Contents

- Azure Architecture Center
- Browse all Architectures
- Architecture icons
- What's new
- Application architecture fundamentals
  - Introduction
  - Architecture styles
    - Overview
    - Big compute
    - Big data
    - Event-driven architecture
    - Microservices
    - N-tier application
    - Web-queue-worker
  - Design principles for Azure applications
    - Overview
    - Design for self-healing
    - Make all things redundant
    - Minimize coordination
    - Design to scale out
    - Partition around limits
    - Design for operations
    - Use managed services
    - Use the best data store for the job
    - Design for evolution
    - Build for the needs of business
  - Technology choices
    - Choose a compute service
    - Choose a microservices compute option

- Choose a Kubernetes edge option
  - Kubernetes at the edge
  - Bare-metal Kubernetes at the edge
- Choose an identity service
- Choose a data store
  - Understand data store models
  - Select a data store
  - Criteria for choosing a data store
- Choose an analytics solution
  - Analytical data stores
  - Analytics and reporting
- Choose an AI/ML service
  - Cognitive services
  - Machine learning
  - Natural language processing
- Choose a load balancing service
- Choose a messaging service
- Choose an IoT solution

Best practices for cloud applications

- Overview
- API design
- API implementation
- Autoscaling
- Background jobs
- Caching
- Content Delivery Network
- Data partitioning
- Data partitioning strategies (by service)
- Message encoding considerations
- Monitoring and diagnostics
- Retry guidance for specific services
- Transient fault handling

- Performance tuning
  - Introduction
  - Scenario 1 - Distributed transactions
  - Scenario 2 - Multiple backend services
  - Scenario 3 - Event streaming
  - Performance antipatterns
    - Overview
    - Busy Database
    - Busy Front End
    - Chatty I/O
    - Extraneous Fetching
    - Improper Instantiation
    - Monolithic Persistence
    - No Caching
    - Noisy Neighbor
    - Retry Storm
    - Synchronous I/O
- Responsible innovation
  - Overview
  - Judgment call
  - Harms modeling
    - Understand harm
    - Assess types of harm
  - Community jury
- Architecture for startups
  - Overview
  - Core startup stack architecture
- Multitenant applications
  - Considerations
    - Overview
    - Tenancy models
    - Tenant lifecycle

- Pricing models
- Measure consumption
- Deploy updates
- Map requests to tenants
- Domain names
- Approaches
- Storage and data
- Service-specific guidance
- Storage and data
- Azure Storage
- Azure SQL Database
- Azure Cosmos DB
- Related resources
- Azure for AWS Professionals
- Overview
- Component information
- Accounts
- Compute
- Databases
- Messaging
- Networking
- Regions and zones
- Resources
- Security and identity
- Storage
- Service comparison
- Azure for GCP Professionals
- Overview
- Services comparison
- Design Patterns
- Overview
- Categories

- Data management
- Design and implementation
- Messaging

Ambassador

Anti-corruption Layer

Asynchronous Request-Reply

Backends for Frontends

Bulkhead

Cache-Aside

Choreography

Circuit Breaker

Claim Check

Compensating Transaction

Competing Consumers

Compute Resource Consolidation

CQRS

Deployment Stamps

Event Sourcing

External Configuration Store

Federated Identity

Gatekeeper

Gateway Aggregation

Gateway Offloading

Gateway Routing

Geode

Health Endpoint Monitoring

Index Table

Leader Election

Materialized View

Pipes and Filters

Priority Queue

Publisher/Subscriber

- Queue-Based Load Leveling
- Rate Limiting
- Retry
- Saga
- Scheduler Agent Supervisor
- Sequential Convoy
- Sharding
- Sidecar
- Static Content Hosting
- Strangler Fig
- Throttling
- Valet Key

Microsoft Azure Well-Architected Framework

- Overview
- Reliability
  - About
  - Overview
  - Principles
  - Design
    - Checklist
    - Requirements
    - Application design
    - Resiliency & dependencies
    - Best practices
  - Testing
    - Checklist
    - Resiliency testing
    - Backup & recovery
      - Backup and recovery
      - Automatic retry of failed backup jobs
    - Error handling
    - Chaos engineering

- Best practices
- Monitoring
  - Checklist
  - Application health
  - Health modeling
  - Best practices
- Reliability patterns
- Security
- Quick links
- About
- Principles
- Design
  - Governance
    - Checklist
    - Compliance requirements
    - Landing zone
    - Segmentation strategy
    - Management groups
    - Administration
  - Identity and access management
    - Checklist
    - Roles and responsibilities
    - Control plane
    - Authentication
    - Authorization
    - Best practices
  - Networking
    - Checklist
    - Network segmentation
    - Connectivity
    - Application endpoints
    - Data flow

- Best practices
- Data protection
  - Checklist
  - Encryption
  - Key and secret management
  - Best practices
- Applications and services
  - Application security considerations
  - Application classification
  - Threat analysis
  - Secure PaaS deployments
  - Configuration and dependencies
- Build-deploy
  - Checklist
  - Governance considerations
  - Infrastructure provisioning
  - Code deployments
- Monitor-remediate
  - Checklist
  - Tools
  - Azure resources
  - Logs and alerts
  - Review and remediate
  - Compliance review
  - Validate and test
  - Security operations
- Tradeoffs

Cost Optimization

- About
- Principles
- Design
  - Checklist

- Cost model
- Capture requirements
- Azure regions
- Azure resources
- Governance
- Initial estimate
- Managed services
- Performance and price options
- Provision
  - Checklist
  - AI + Machine Learning
  - Big data
  - Compute
  - Data stores
  - Messaging
  - Networking
    - Cost for networking services
  - Web apps
- Monitor
  - Checklist
  - Budgets and alerts
  - Reports
  - Reviews
- Optimize
  - Checklist
  - Autoscale
  - Reserved instances
  - VM instances
  - Caching
- Tradeoffs

Operational Excellence

- About

- Overview
- Principles
- Automation
  - Automation Overview
  - Repeatable infrastructure
  - Configure infrastructure
  - Automate operational tasks
- Release engineering
  - Application development
  - Continuous integration
  - Release testing
  - Performance
  - Release deployment
  - Rollback
- Monitor and alerts
  - Monitoring
  - Alerting
- Checklist
- Operational Excellence patterns

Performance Efficiency

- About
- Overview
- Principles
- Design
  - Checklist
  - Distributed architecture challenges
  - Application design
  - Application efficiency
  - Scalability
  - Capacity planning
- Test
  - Checklist

- Performance testing
- Testing tools
- Monitoring
- Performance Efficiency patterns
- Checklist
- Tradeoffs
- Workloads
- Hybrid
- Overview
- Cost Optimization
- Operational Excellence
- Performance Efficiency
- Reliability
- Security
- Services
- Compute
- Azure App Service
- Reliability
- Cost optimization
- Operational excellence
- Azure Batch
- Reliability
- Operational excellence
- Performance efficiency
- Azure Kubernetes Service
- Reliability
- Security
- Cost optimization
- Operational excellence
- Performance efficiency
- Functions
- Security

- Service Fabric
  - Reliability
  - Security
  - Operational excellence
  - Performance efficiency
- Virtual Machines
  - Reliability
  - Cost optimization
  - Operational excellence
- Data
  - Azure Cache for Redis
    - Reliability
    - Operational excellence
  - Azure Databricks
    - Security
  - Azure Database for MySQL
    - Cost optimization
  - Azure Database for PostgreSQL
    - Cost optimization
  - Azure SQL Database
    - Reliability
    - Cost optimization
    - Operational excellence
  - Azure SQL Managed Instance
    - Reliability
    - Operational excellence
  - Cosmos DB
    - Reliability
    - Operational excellence
- Industry solutions with Azure
  - Retail
    - Overview

- Guides
  - Data management in the retail industry
  - Migrate your e-commerce solution to Azure
  - Optimize and reuse recommendations
  - Visual search in retail with CosmosDB
  - SKU optimization for consumer brands
- Architectures
  - Build a real-time recommendation API
  - Content-based recommendation
  - E-commerce front end
  - Hybrid AI footfall detection for retail
  - Intelligent search engine for e-commerce
  - Magento e-commerce platform in AKS
  - Movie recommendations on Azure
  - Retail - Buy online, pickup in store
  - Scalable order processing
- Finance
  - Overview
  - Guides
    - Risk grid computing in banking
    - Risk grid computing solution
    - Data management in banking
    - Actuarial risk analysis
    - Enable the financial services risk lifecycle
  - Architectures
    - Banking system cloud transformation
    - Decentralized trust between banks
    - Modernize mainframe & midrange data
    - Patterns and implementations in banking
    - Real-time fraud detection
    - Replicate and sync mainframe data in Azure
  - Compliance architecture

- AKS regulated cluster for PCI
- Blueprint - Analytics for PCI DSS
- Blueprint - Data warehouse for PCI DSS
- Blueprint - IaaS web application for PCI DSS
- Blueprint - PaaS web application for PCI DSS

Healthcare

- Overview
- Guides
  - Microsoft Cloud for Healthcare
    - Overview
    - Patient engagement
    - Health team collaboration
    - Clinical and operational insights
    - Security in Cloud for Healthcare
  - Healthcare APIs
    - Overview
    - Healthcare APIs workspace
  - Text Analytics for health
    - Overview
    - Recognized entity categories
    - Relation extraction
    - Assertion detection
- Architectures
  - Build a telehealth system with Azure
  - Clinical insights with Cloud for Healthcare
  - Confidential computing for healthcare
  - Consumer health portal on Azure
  - Health data consortium
  - Population health management
  - Predict hospital readmissions with ML
  - Precision medicine pipeline
  - Virtual visits with Cloud for Healthcare

- Compliance architecture
  - Blueprint - HIPAA/HITRUST data and AI
  - Implement the healthcare blueprint for AI
  - HIPAA and HITRUST compliant health data
- Government
  - Overview
  - Guides
    - Compare Azure Government and Azure
    - Considerations for naming resources
    - Development
      - Azure Government developer guide
      - Storage on Azure Government
      - AI on Azure Government
      - SSMS on Azure Government
    - Security
      - Security for Azure Government
      - Impact Level 5 isolation
      - Secure isolation
      - Secure Azure computing
    - Compliance
    - Identity
      - Identity for Azure Government
      - Integrate Azure AD authentication
    - Deployment
      - Deploy with Azure Pipelines
      - ASE with DISA CAP
    - Management
      - Azure Monitor logs
      - Marketplace
  - Architectures
    - Azure Automation in a hybrid environment
    - Azure Automation update management

- Computer forensics chain of custody in Azure
- Hybrid security monitoring in Azure
- Web app private database connectivity
- Azure Virtual Desktop for the enterprise
- Compliance architecture
  - Azure FedRAMP compliance
  - Blueprint - Analytics for FedRAMP
  - Blueprint - Data warehouse for FedRAMP
  - Blueprint - IaaS web app for FedRAMP
  - Blueprint - PaaS web app for FedRAMP
  - FedRAMP high compliance
  - FedRAMP moderate compliance
- Manufacturing
  - Overview
  - Guides
    - Industrial IoT analytics
    - Upscale ML lifecycle with MLOps framework
    - On-demand, scalable, high-power compute
    - Predictive maintenance in manufacturing
    - Predictive maintenance solution
    - Extract actionable insights from IoT data
  - Architectures
    - Anomaly detector process
    - Build a speech-to-text transcription pipeline
    - Computer vision on the edge
    - Predictive maintenance with IoT
    - Supply chain track and trace
- Media and entertainment
  - Overview
  - 3D video rendering
  - Content-based recommendation
  - Digital image-based modeling on Azure

- Image classification on Azure
- Movie recommendations on Azure
- Energy and environment
  - Run reservoir simulations
  - Geospatial data processing and analytics
  - Run CFD simulations
  - Project 15 sustainability
  - Environmental monitoring
  - Oil and gas tank level forecasting
- Game development
  - Rendering
    - 3D video rendering
    - Digital image-based modeling on Azure
  - Databases for gaming
    - Gaming using Azure MySQL
    - Gaming using Cosmos DB
  - Game server hosting
    - Basic game server hosting
    - LAMP architectures for gaming
  - Scalable gaming servers
    - Multiplayer backend architectures
    - Real-time multiplayer
      - Multiplayer server hosting with Batch
      - Multiplayer server hosting with AKS
      - Multiplayer hosting with Service Fabric
      - Multiplayer server hosting with ACI
      - Custom game server scaling
    - Turn-based multiplayer
      - Asynchronous multiplayer basics
      - Serverless asynchronous multiplayer
  - Game analytics
    - Non-real-time dashboard

- In-editor debugging telemetry
- Matchmaking
  - Multiplayer matchmaker
  - Serverless matchmaker
- Leaderboards
  - Leaderboard basics
  - Non-relational leaderboard
  - Relational leaderboard
- AI in games
  - Content moderation
  - Customer service bot for gaming
  - Image classification
  - Speech to text for gaming
  - Text to speech for gaming
  - Text translation for gaming
- Game streaming
  - Unreal Pixel Streaming
  - Deploy Unreal Pixel Streaming
  - Unreal Pixel Streaming at scale
- Automotive and transportation
  - Automated guided vehicles fleet control
  - Building blocks for autonomous driving
  - Predictive insights with vehicle telematics
  - Process vehicle data using IoT
  - Run CFD simulations
- Azure categories
  - AI + Machine Learning
    - Get started
    - Guides
      - Architecture and key concepts
      - Cognitive services
      - Machine learning

ML architecture and key concepts
Machine learning at scale
Security baseline for Azure Machine Learning
Natural language processing
R developer's guide to Azure
Machine teaching
MLOps framework
  Upscale ML lifecycle with MLOps
  MLOps maturity model
  Azure ML service selection guide
Industry guidance
  Enable the financial services risk lifecycle
  Healthcare blueprint for AI
  Optimize and reuse recommendations
  Predictive maintenance in manufacturing
  Predictive maintenance solution
  SKU optimization for consumer brands
  Visual search for retail
Team Data Science Process
  Overview
  Lifecycle
    Overview
    1. Business understanding
    2. Data acquisition and understanding
    3. Modeling
    4. Deployment
    5. Customer acceptance
  Roles and tasks
    Overview
    Group manager
    Team lead
    Project lead

- Individual contributor
- Project planning
- Development
  - Agile development
  - Collaborative coding with Git
  - Execute data science tasks
  - Code testing
  - Track progress
- Operationalization
  - DevOps - CI/CD
- Worked-out examples
  - Overview
  - Spark examples
  - Hive with HDInsight Hadoop
  - U-SQL with Azure Data Lake
  - R, Python and T-SQL with SQL Server
  - T-SQL and Python with Synapse Analytics
- Training
  - For data scientists
  - For DevOps
- How To
  - Set up data science environments
  - Analyze business needs
  - Acquire and understand data
  - Develop models
  - Deploy models in production
- Related
  - Anomaly detection
  - Predictive maintenance

Architectures

- Baseball decision analysis with ML.NET
- Batch scoring for deep learning

- Batch scoring with Python
- Batch scoring with R
- Batch scoring with Spark on Databricks
- Blueprint architectures
  - Blueprint - HIPAA/HITRUST health data and AI
  - Implement the healthcare blueprint for AI
- Chatbot for hotel booking
- Citizen AI with the Power Platform
- Computer vision on the edge
- Deploy AI and ML at the edge
- Determine customer lifetime and churn
- Distributed deep learning training
- Enterprise-grade conversational bot
- Image classification
- MLOps for Python models
- Movie recommendations
- Orchestrate MLOps on Azure Databricks
- Predict hospital readmissions with ML
- Real-time recommendation API
- Real-time scoring Python models
- Real-time scoring R models
- Scalable personalization
- Scale AI and ML in regulated industries
- Speech to text conversion
- Training Python models
- Solution ideas
  - AI at the edge
  - AI enrichment with Cognitive Search
  - Auditing and risk management
  - Autonomous systems
  - Business process management
  - Content research

Content tagging with NLP

Contract management

Customer churn prediction

Customer feedback

Data science and machine learning

Defect prevention

Digital asset management

Disconnected AI at the edge

E-commerce chatbot

Energy demand forecasting

Energy supply optimization

Enterprise chatbot disaster recovery

Enterprise productivity chatbot

Environment monitoring

FAQ chatbot

Hospital patient predictions

Image classification with CNNs

Interactive voice response bot

Keyword digital text processing

Marketing optimization

Model deployment to AKS

Personalized marketing solutions

Personalized offers

Population health management

Predictive maintenance

Predictive marketing

Quality assurance

Remote patient monitoring

Retail assistant with visual capabilities

Retail product recommendations

Secure research for regulated data

Speech services

- Vehicle telematics
- Vision classifier model
- Visual assistant
- Analytics
  - Get started
  - Guides
    - Technology choices
      - Analytical data stores
      - Analytics and reporting
      - Batch processing
    - Actuarial risk analysis
    - Analytics security baselines
      - Security baseline for Azure Databricks
      - Security baseline for Azure Data Factory
      - Security baseline for Azure Purview
    - Apache NiFi guidance
      - Apache NiFi monitoring with MonitoFi
      - Helm-based deployments for Apache NiFi
  - Architectures
    - Anomaly detector process
    - Apache NiFi on Azure
    - Automated enterprise BI
    - Azure analytics end to end
    - Blueprint architectures
      - Blueprint - Analytics for FedRAMP
      - Blueprint - Analytics for PCI DSS
    - Data warehouse for small business
    - Data warehousing and analytics
    - Geospatial data processing and analytics
    - High throughput stream ingestion
    - Ingestion and analysis of news feeds
    - IoT and data analytics

- Long-term security logs in Azure Data Explorer
- Partitioning in Event Hubs and Kafka
- Precision medicine pipeline with genomics
- Stream processing with Azure Databricks
- Stream processing with Stream Analytics
- Stream processing with fully managed open-source data engines
- Solution ideas
  - Advanced analytics
  - App integration using Event Grid
  - Big data analytics with Azure Data Explorer
  - Big data analytics with enterprise security
  - Content Delivery Network analytics
  - Demand forecasting for shipping
  - Demand forecasting and price optimization
  - Demand forecasting with Stream Analytics
  - Discovery Hub for analytics
  - ETL using HDInsight
  - Highly scalable customer service and ERP
  - Hybrid big data with HDInsight
  - Ingestion, ETL, and stream processing
  - Interactive analytics with Data Explorer
  - Interactive price analytics
  - IoT analytics with Azure Data Explorer
  - Mining equipment monitoring
  - Modern analytics with Azure Databricks
  - Monitoring solution with Data Explorer
  - Oil and Gas tank level forecasting
  - Predicting length of stay in hospitals
  - Predictive aircraft engine monitoring
  - Real Time analytics on big data
  - Tiering applications & data for analytics
- Blockchain + Multiparty Compute

- Guides
  - Azure Confidential Ledger architecture
  - Authenticate Confidential Ledger nodes
  - Multiparty computing
- Architectures
  - Azure SQL Database ledger
  - Decentralized trust between banks
  - Multi-cloud blockchain DLT
- Solution ideas
  - Blockchain workflow application
  - Supply chain track and trace
- Compute
  - Get started with HPC
  - Guides
    - Choose a compute service
    - SAS on Azure architecture
    - Industry guidance
      - High-power compute for manufacturing
      - Risk grid computing in banking
      - Risk grid computing solution
    - VM security baselines
      - Security baseline for virtual machine scale sets
      - Security baseline for Linux VMs
      - Security baseline for Windows VMs
  - Architectures
    - 3D video rendering
    - Blueprint architectures
      - Blueprint - IaaS web application for FedRAMP
      - Blueprint - IaaS web application for PCI DSS
    - Computer-aided engineering
    - Digital image modeling
    - Linux virtual desktops with Citrix

- Move Azure resources across regions
- Run a Linux VM on Azure
- Run a Windows VM on Azure
- Run CFD simulations
- Run reservoir simulations
- Solution ideas
  - HPC cluster deployed in the cloud
  - HPC risk analysis
  - HPC system and big compute
  - Hybrid HPC with HPC Pack
- Containers
  - Get started
  - Guides
    - AKS cluster best practices
    - AKS day-2 operations guide
      - Introduction
      - Triage practices
        - Overview
        - 1- Cluster health
        - 2- Node and pod health
        - 3- Workload deployments
        - 4- Admission controllers
        - 5- Container registry connectivity
      - Patching and upgrade guidance
      - Monitoring with Azure Monitor
      - Common issues
    - Choose a Kubernetes option
      - Choose a Kubernetes at the edge option
      - Choose a bare-metal Kubernetes option
    - Governance disciplines for AKS
    - Cost governance with Kubecost
    - BCDR considerations for AKS

- Security baseline for AKS
- Architectures
  - Advanced microservices architecture on AKS
  - AKS baseline cluster
  - AKS baseline for multi-region clusters
  - AKS cluster for a PCI-DSS workload
    - Introduction
    - Architecture
    - Network segmentation
    - Data protection
    - Vulnerability management
    - Access controls
    - Monitoring operations
    - Policy management
    - Summary
  - Autonomous-driving simulation
  - Building a telehealth system
  - CI/CD pipeline for container-based workloads
  - GitOps for AKS
  - Magento e-commerce in AKS
  - Microservices architecture on AKS
  - Multiplayer server hosting with ACI
  - Multiplayer server hosting with AKS
  - Multiplayer hosting with Service Fabric
  - Multitenancy with AKS and AGIC
- Solution ideas
  - API-first SaaS business model with AKS
  - Azure Kubernetes in event stream processing
  - Bursting from AKS with ACI
  - Data streaming with AKS
  - Elastic demand handling with AKS
  - Instant IoT data streaming with AKS

- JBoss deployment with Red Hat on Azure
- Microservices with AKS and Azure DevOps
- Secure DevOps for AKS
- Databases
  - Guides
    - Get started
    - Relational data
      - Extract, transform, and load (ETL)
      - Online analytical processing (OLAP)
      - Online transaction processing (OLTP)
      - Data Warehousing
    - Non-relational data
      - Non-relational data stores
      - Free-form text search
      - Time series data
      - Working with CSV and JSON files
    - Build a scalable system for massive data
    - Big Data
      - Big Data architectures
      - Batch processing
      - Real time processing
    - Technology choices
      - Analytical data stores
      - Analytics and reporting
      - Batch processing
      - Data lakes
      - Data storage
      - Choose a data store
        - Understand data store models
        - Select a data store
        - Criteria for choosing a data store
      - Pipeline orchestration

- Real-time message ingestion
- Search data stores
- Stream processing
- Data management patterns
- Use the Transactional Outbox pattern
- Industry guidance
  - Data management in banking
  - Data management in retail
  - Visual search for retail
- Transfer data to and from Azure
- Extend on-premises data solutions to Azure
- Secure data solutions
- Tenancy models for SaaS databases
- Monitor Azure Databricks jobs
  - Overview
  - Send Databricks application logs to Azure Monitor
  - Use dashboards to visualize Databricks metrics
  - Troubleshoot performance bottlenecks
  - Observability patterns and metrics for performance tuning
- Run Apache Cassandra

Architectures

- Azure health data consortium
- Blueprint architectures
  - Blueprint - Data warehouse for FedRAMP
  - Blueprint - Data warehouse for PCI DSS
  - Blueprint - HIPAA/HITRUST health data and AI
- Cost savings through HTAP with Azure SQL
- Data governance with Profisee
- DataOps for modern data warehouse
- Hybrid ETL with Azure Data Factory
- Master data management with CluedIn
- Master data management with Profisee

Migrate master data services with CluedIn
N-tier app with Cassandra
SQL 2008 R2 failover cluster in Azure
SQL Database and Synapse Analytics connectivity
Windows N-tier applications
Solution ideas
Big data analytics with Azure Data Explorer
Campaign optimization with HDInsight Spark
Campaign optimization with SQL Server
Data streaming
Data cache
Digital campaign management
Digital marketing using Azure MySQL
Enterprise data warehouse
Finance management using Azure MySQL
Finance management using Azure PostgreSQL
Gaming using Azure MySQL
Gaming using Cosmos DB
Globally distributed apps using Cosmos DB
Intelligent apps using Azure MySQL
Intelligent apps using Azure PostgreSQL
Interactive querying with HDInsight
Loan charge-off prediction with HDInsight Spark
Loan charge-off prediction with SQL Server
Loan credit risk modeling
Loan credit risk with SQL Server
Messaging
Minimal storage – change feed to replicate data
Mining equipment monitoring
Multi-region web application with Cosmos DB replication
Ops automation using Event Grid
Optimized storage with logical data classification

- Optimized storage – time based - multi writes
- Optimized storage – time based with Data Lake
- Oracle migration to Azure
- Personalization using Cosmos DB
- Retail and e-commerce using Azure MySQL
- Retail and e-commerce using Azure PostgreSQL
- Retail and e-commerce using Cosmos DB
- Running Oracle Databases on Azure
- Serverless apps using Cosmos DB
- Streaming using HDInsight

Developer Options

- Microservices
  - Get started
  - Guides
    - Domain modeling for microservices
      - Domain analysis
      - Tactical DDD
      - Identify microservice boundaries
    - Design a microservices architecture
      - Introduction
      - Choose a compute option
      - Interservice communication
      - API design
      - API gateways
      - Data considerations
      - Container orchestration
      - Design patterns for microservices
    - Operate microservices in production
      - Monitor microservices in Azure Kubernetes Service (AKS)
      - CI/CD for microservices
      - CI/CD for microservices on Kubernetes
    - Migrate to a microservices architecture

- Migrate a monolith application to microservices
- Modernize enterprise applications with Service Fabric
- Migrate from Cloud Services to Service Fabric
- Architectures
  - Decompose apps with Service Fabric
  - Microservices on Azure Service Fabric
  - Microservices with Azure Spring Cloud
  - Unified logging for microservices apps
- Serverless applications
  - Get started
  - Guides
    - Serverless Functions examples
    - Plan for serverless architecture
      - Serverless Functions decision and planning
      - Serverless application assessment
      - Technical workshops and training
      - Proof of concept or pilot
    - Develop and deploy serverless apps
      - Serverless Functions app development
      - Serverless Functions code walkthrough
      - CI/CD for a serverless frontend
    - Monitoring serverless event processing
    - Serverless Functions app operations
    - Serverless Functions app security
    - Security baseline for Azure Functions
    - Serverless with Azure Logic Apps
    - Event Hubs with Azure Functions
      - Overview
      - Performance and scale
      - Resilient design
      - Security
      - Observability

- Architectures
  - Azure Functions in a hybrid environment
  - Event-based cloud automation
  - Multicloud with the Serverless Framework
  - Real-time location sharing
  - Serverless event processing
- Solution ideas
  - Azure Kubernetes in event stream processing
  - Big data analytics with Azure Data Explorer
  - De-batching and filtering in serverless event processing with Event Hubs
  - Private link scenario in event stream processing
  - Serverless applications using Event Grid
  - Serverless computing LOB apps
  - Transit Hub pub-sub messaging system

DevOps

- Guides
  - DevOps Checklist
  - Advanced Azure Resource Manager templates
    - Overview
    - Update a resource
    - Use an object as a parameter
    - Property transformer and collector
  - DevTest Labs guidance
  - Continuous integration for Synapse Analytics
  - Platform automation for VMware Solution
- Architectures
  - Azure DevTest Labs for enterprises
  - CI/CD pipeline for chatbots with ARM
  - CI/CD pipeline using Azure DevOps
  - DevSecOps in GitHub
  - Enterprise monitoring with Azure Monitor
  - Jenkins on Azure

- Run containers in a hybrid environment
- Solution ideas
  - CI/CD for Azure VMs
  - CI/CD for Azure Web Apps
  - CI/CD for Containers
  - CI/CD for Microsoft Power Platform
  - CI/CD using Jenkins and AKS
  - CI/CD using Jenkins and Terraform
  - DevSecOps in Azure
  - DevTest and DevOps for IaaS
  - DevTest and DevOps for PaaS
  - DevTest and DevOps with microservices
  - DevTest Image Factory
  - Hybrid DevOps
  - Java CI/CD using Jenkins and Web Apps
  - SharePoint for development testing
- High Availability + Disaster Recovery
  - Guides
    - Build solutions with Availability Zones
    - High availability for SQL DB and Managed Instance
  - Architectures
    - AKS baseline for multi-region clusters
    - Highly available multi-region web app
    - IaaS: Web app with relational database
    - Multi-region app with private database
    - Multi-region load balancing
    - Multi-region N-tier application
    - Multitenant SaaS
    - Multi-tier web application built for HA/DR
- Hybrid + Multicloud
  - Get started
  - Guides

- Hybrid workload in Azure
- Azure Arc guidance
- Hybrid deployments
  - Configure hybrid cloud connectivity in Azure and Azure Stack Hub
  - Configure hybrid cloud identity for Azure and Azure Stack Hub apps
  - Deploy AI-based footfall detection solution in Azure and Azure Stack Hub
  - Deploy an app that scales cross-cloud in Azure and Azure Stack Hub
  - Deploy highly available Kubernetes cluster on Azure Stack Hub
  - Deploy a highly available MongoDB solution to Azure and Azure Stack Hub
  - Deploy hybrid app with on-premises data that scales cross-cloud
  - Deploy a SQL Server 2016 availability group to Azure and Azure Stack Hub
  - Direct traffic with a geo-distributed app using Azure and Azure Stack Hub
- Azure Stack stretched clusters for DR
- Azure Stack for remote offices and branches
- Azure VMware Solution guidance
  - API Management for Azure VMware Solution
  - BCDR for Azure VMware Solution
  - Govern Azure VMware Solution
  - Manage and monitor Azure VMware Solution
  - Migrate with Azure VMware Solution
  - Network interconnectivity for VMware Solution
  - Network planning for Azure VMware Solution
  - Platform automation for Azure VMware Solution
  - Security and governance for VMware Solution
- Connect an on-premises network to Azure
- FSLogix for the enterprise
- Troubleshoot a hybrid VPN connection

Architectures

- Azure Arc solutions
  - Azure Arc hybrid management with AKS
  - Manage configurations for Azure Arc
- Azure Automation solutions

- Azure Automation in a hybrid environment
- Azure Automation update management
- Azure Automation State Configuration
- Azure enterprise cloud file share
- Azure Functions in a hybrid environment
- Azure VMware Solution in hub-and-spoke
- Back up files on Azure Stack Hub
- Connect on-premises with ExpressRoute
- Connect standalone servers with Network Adapter
- Deploy AI and ML with Azure Stack Edge
- Design a hybrid Domain Name System
- Disaster recovery for Stack Hub VMs
- Extend on-premises using ExpressRoute
- Extend an on-premises network using VPN
- Hybrid availability and monitoring
- Hybrid file services
- Hybrid file share with disaster recovery for remote and local branch workers
- Hybrid security monitoring
- Manage hybrid workloads with WAC
- On-premises data gateway for Logic Apps
- Run containers in a hybrid environment
- Azure file shares in hybrid environment
- Solution ideas
- Azure VMware Solution foundations
- Azure VMware Solution capacity planning
- Azure VMware Solution landing zone
- Azure VMware solution networking
- Cross-cloud scaling
- Cross-platform chat
- Hybrid connections
- Unlock legacy data with Azure Stack
- Identity

- Guides
  - Compare identity services
  - Build for resilience
  - Identity in multitenant applications
    - Introduction
    - The Tailspin scenario
    - Authentication
    - Claims-based identity
    - Tenant sign-up
    - Application roles
    - Authorization
    - Secure a web API
    - Cache access tokens
    - Client certificate
    - Federate with a customer's AD FS
  - Integrate on-premises AD with Azure
  - Migrate applications to Azure AD
    - Migrate an AD FS app to Azure
    - Migrate app authentication to Azure AD
    - Use the AD FS application activity report
    - Resources for migrating to Azure AD
  - Secure development with SPAs
- Architectures
  - AD DS resource forests in Azure
  - Azure AD identity management for AWS
  - Deploy AD DS in an Azure virtual network
  - Extend on-premises AD FS to Azure
  - Hybrid identity
  - On-premises AD domains with Azure AD
- Integration
  - Guides
    - Connectors in Azure Logic Apps

- Access to virtual networks from Logic Apps
- BCDR for Logic Apps
- Azure Policy controls for Logic Apps
- Security baseline for Logic Apps
- Architectures
  - Basic enterprise integration on Azure
  - Data integration with Logic Apps and SQL
  - Enterprise business intelligence
  - Enterprise integration - queues and events
  - On-premises data gateway for Logic Apps
  - Power Automate deployment at scale
  - Publish internal APIs to external users
- Solution ideas
  - Custom business processes
  - Elastic Workplace Search on Azure
  - Line of business extension
  - Web and mobile front-ends
- Internet of Things
  - Get started
  - Guides
    - IoT concepts
      - Introduction to IoT solutions
      - Devices, platform, and applications
      - Attestation, authentication, and provisioning
      - Field and cloud edge gateways
      - Application-to-device commands
      - Builders, developers, and operators
    - Azure IoT Edge Vision
      - Overview
      - Camera selection
      - Hardware acceleration
      - Machine learning

- Image storage
- Alert persistence
- User interface
- Industrial IoT Analytics
  - Architecture
  - Recommended services
  - Data visualization
  - Considerations
- IoT patterns
  - Analyze and optimize loop
  - Event routing
  - Measure and control loop
  - Monitor and manage loop
  - Real-time IoT updates
  - Scale solutions with deployment stamps
- Azure IoT client SDK support
- Choose an IoT solution
- Moving from test to production

Architectures

- Azure IoT reference architecture
- Automated guided vehicles fleet control
- Buy online, pick up in store
- Computer vision on the edge
- Connected factory Hierarchy Service
- Efficient Docker image deployment
- IoT and data analytics
- IoT using Cosmos DB
- Predictive maintenance with IoT
- Project 15 IoT sustainability
- Smart places with Azure Digital Twins

Solution ideas

- Condition monitoring

- COVID-19 solutions
  - Cognizant Safe Buildings with IoT
  - Contactless IoT interfaces
  - COVID-19 IoT safe environments
  - IoT Connected Platform
  - Lighting and disinfection system
- Environment monitoring
- IoT analytics with Azure Data Explorer
- IoT Edge data storage and processing
- Light and power for emerging markets
- Process real-time vehicle data using IoT
- Real-time asset tracking and management
- Voice assistants and IoT devices

Mainframe + Midrange

- Get started
- Guides
  - Mainframe migration framework
    - Mainframe migration overview
    - Mainframe myths and facts
    - Mainframe migration strategies
    - Mainframe application migration
  - Mainframe rehosting
    - Mainframe rehosting on Azure virtual machines
    - Move mainframe compute to Azure
    - Move mainframe storage to Azure
    - Get started with TmaxSoft OpenFrame on Azure
- App modernization
  - Architectures
    - AIX UNIX to Azure Linux migration
    - Batch transaction processing
    - General mainframe refactor to Azure
    - IBM System i to Azure using Infinite i

- IBM z/OS migration with Asysco AMT
- IBM z/OS online transaction processing
- Integrate IBM MQs with Azure
- Micro Focus Enterprise Server on Azure
- Migrate AIX workloads to Skytap
- Migrate IBM i series to Skytap on Azure
- Refactor IBM z/OS mainframe CF
- Refactor Mainframe Applications with Advanced
- Refactor mainframe with Raincode
- Unisys CPF rehost using virtualization
- Unisys Dorado migration
- Unisys mainframe migration with Asysco
- Using LzLabs SDM in Azure
- Solution ideas
  - Migrate IBM apps with TmaxSoft
  - Solaris emulator on Azure VMs
- Data modernization
  - Architectures
    - Mainframe and midrange data replication to Azure using Qlik
    - Modernize mainframe and midrange data
    - Re-engineer mainframe batch apps
    - Replicate and sync mainframe data
  - Solution ideas
    - Mainframe access to Azure databases
    - Mainframe file replication on Azure
- Management + Governance
  - Guides
    - Backup
      - Backup architecture and components
      - Azure Backup support matrix
      - Backup cloud and on-premises workloads
    - Disaster recovery

- Azure to Azure disaster recovery
- Support matrix for Azure VM disaster recovery
- ExpressRoute with Azure VM disaster recovery
- Management and monitoring
- Management for VMware Solution
- Architectures
  - Back up cloud applications
  - Computer forensics
  - End-to-end governance when using CI/CD
  - Highly available SharePoint Server 2016
  - Updating Windows VMs in Azure
  - Hybrid management
    - Azure Arc hybrid management and deployment for Kubernetes clusters
    - Azure Automation in a hybrid environment
    - Azure Automation Update Management
    - Back up files and applications on Azure Stack Hub
    - Disaster Recovery for Azure Stack Hub virtual machines
    - Hybrid availability and performance monitoring
    - Manage configurations for Azure Arc enabled servers
    - Manage hybrid Azure workloads using Windows Admin Center
  - Solution ideas
    - Archive on-premises data to cloud
    - Back up on-premises applications
    - Centralize app configuration and security
    - Data Sovereignty & Data Gravity
    - Enterprise-scale disaster recovery
    - High availability for BCDR
    - SMB disaster recovery with Azure Site Recovery
    - SMB disaster recovery with Double-Take DR
- Media
  - Guides
    - Media Services terminology and concepts

- Encoding video and audio with Media Services
- Security baseline for Azure Media Services
- Architectures
  - Gridwich media processing system
    - Gridwich architecture
    - Gridwich concepts
      - Clean monolith design
      - Saga orchestration
      - Project names and structure
      - Gridwich CI/CD
      - Content protection and DRM
      - Gridwich Media Services
      - Gridwich Storage Service
      - Gridwich logging
      - Gridwich message formats
      - Pipeline variables to Terraform flow
    - Gridwich procedures
      - Set up Azure DevOps
      - Run Azure admin scripts
      - Set up local dev environment
      - Create new cloud environment
      - Maintain and rotate keys
      - Test Media Services V3 encoding
  - Solution ideas
    - Instant broadcasting with serverless
    - Live streaming digital media
    - Video-on-demand digital media
- Migration
  - Guides
    - Migrate an e-commerce solution to Azure
    - Migrate with Azure VMware Solution
  - Architectures

- Banking system
  - Banking cloud transformation
  - Patterns and implementations
  - JMeter implementation reference
- Lift and shift LOB apps
- Oracle database migration
  - Migration decision process
  - Cross-cloud connectivity
  - Lift and shift to Azure VMs
  - Refactor
  - Rearchitect
- Mixed Reality
  - Guides
    - Design and prototype for mixed reality
    - MR core concepts
      - App model
      - App views
      - Audio in mixed reality
      - Comfort
      - Coordinate systems
      - Eye tracking
      - Holographic frame
      - Room scan visualization
      - Scene understanding
      - Spatial anchors
      - Spatial mapping
      - Start gesture
    - Types of mixed reality apps
    - Design content for holographic display
    - Shared experiences in mixed reality
  - Solution ideas
    - Design review with mixed reality

- Facilities management with mixed reality
- Training powered by mixed reality
- Mobile
  - Guides
    - Choose a mobile development framework
    - Add authentication in mobile apps
    - Store, sync, and query mobile app data
    - Cloud storage for mobile apps
  - Solution ideas
    - Adding mobile front-ends to legacy apps
    - Custom mobile workforce app
    - Scalable apps with Azure MySQL
    - Scalable apps using Azure PostgreSQL
    - Social app for with authentication
    - Task-based consumer mobile app
- Networking
  - Guides
    - Add IP spaces to peered virtual networks
    - Application Gateway framework review
    - Azure Firewall architecture overview
    - Azure Firewall framework review
    - NAT gateway framework review
    - Private Link in hub-and-spoke network
    - Segment Virtual Networks
    - Use ExpressRoute with Power Platform
      - Overview
      - Benefits of using ExpressRoute
      - How ExpressRoute works
      - Before you use ExpressRoute
      - Understand Power Platform architecture
      - Plan an ExpressRoute deployment
      - Set up ExpressRoute for Power Platform

- ExpressRoute readiness checklist
- Virtual network peering and VPN gateways
- Architectures
  - Deploy highly available NVAs
  - High availability for IaaS apps
  - Hub-spoke network topology in Azure
  - Hub-spoke topology with Virtual WAN
  - Hybrid networking
    - Azure Automation Update Management
    - Connect servers with Network Adapter
    - Design a hybrid DNS solution
    - Hybrid availability and monitoring
  - Implement a secure hybrid network
- Solution ideas
  - Low-latency network for industry
  - Video capture and analytics for retail
  - IoT network for healthcare facilities
SAP
- Get started
- Guides
  - Checklist
  - SAP HANA infrastructure configurations
  - SAP workload configurations with AZs
  - Supported scenarios for HLI
- Architectures
  - Dev/test for SAP
  - SAP BusinessObjects BI platform
  - SAP BusinessObjects BI platform for Linux
  - SAP BW/4HANA in Linux on Azure
  - SAP deployment using an Oracle DB
  - SAP HANA on HLI general architecture
  - SAP HANA on HLI network architecture

- SAP HANA on HLI with HA and DR
- SAP HANA scale-up on Linux
- SAP NetWeaver on Windows on Azure
- SAP S/4HANA in Linux on Azure
- SAP System on Oracle Database on Azure
- Solution ideas
  - SAP NetWeaver on SQL Server
  - SAP S/4 HANA for Large Instances
  - SAP workload automation using SUSE
- Security
  - Guides
    - Highly-secure IaaS apps
    - Security design principles
    - Virtual Network security options
    - Zero-trust network for web applications
  - Architectures
    - Azure AD in Security Operations
    - Cyber threat intelligence
    - Healthcare platform confidential computing
    - Homomorphic encryption with SEAL
    - Hybrid security monitoring
    - Improved-security access to multitenant web apps from on-premises
    - Long-term security logs in Azure Data Explorer
    - MCAS and Azure Sentinel security for AWS
    - Multi-region app with private database
    - Multilayered protection for Azure VMs
    - Real-time fraud detection
    - Restrict interservice communications
    - Secure OBO refresh tokens
    - Securely managed web apps
    - Secure a Microsoft Teams channel bot
    - Virtual network integrated microservices

- Web app private database connectivity
- Storage
  - Guides
    - Storage accounts
    - Data protection
    - Storage encryption for data at rest
    - Azure Policy controls for Storage
  - Architectures
    - Hybrid file services
    - Oracle Database with Azure NetApp Files
    - SQL Server on Azure Virtual Machines with Azure NetApp Files
    - Azure file shares in hybrid environment
  - Solution ideas
    - HIPAA/HITRUST Health Data and AI
    - Media rendering
    - Medical data storage
    - Multi-region web app with replication
    - Two-region app with Table storage failover
- Virtual Desktop
  - Guides
    - Authentication in Azure Virtual Desktop
    - Azure Virtual Desktop network connectivity
    - Azure AD join for Azure Virtual Desktop
    - Connect RDP Shortpath
    - FSLogix guidance
      - FSLogix for the enterprise
      - FSLogix profile containers and files
      - Storage options for FSLogix profile containers
  - Architectures
    - Azure Virtual Desktop for the enterprise
    - Multiple Active Directory forests
    - Multiple forests with Azure AD DS

- Web
  - Guides
    - Design principles
    - Design and implementation patterns
    - App Service considerations
      - Deployment best practices
      - Security recommendations for App Service
      - Security baseline for App Service
      - Networking features for App Service
  - Architectures
    - Basic web application
    - Blueprint architectures
      - Blueprint - PaaS web application for FedRAMP
      - Blueprint - PaaS web application for PCI DSS
    - Clinical insights with Microsoft Cloud for Healthcare
    - Consumer health portal
    - Deployment in App Service Environments
      - Standard deployment
      - High availability deployment
    - E-commerce front end
    - Improved-security access to multitenant web apps from on-premises
    - Intelligent search engine for e-commerce
    - Migrate a web app using Azure APIM
    - Multi-tier app service with service endpoint
    - Protect APIs with Application Gateway
    - Real-time location sharing
    - Scalable and secure WordPress on Azure
    - Scalable order processing
    - Scalable web app
    - Serverless web app
    - Virtual visits with Microsoft Cloud for Healthcare
    - Web app monitoring on Azure

Solution ideas
Dynamics Business Central as a Service on Azure
E-commerce website running in ASE
Highly available SharePoint farm
Hybrid SharePoint farm with Microsoft 365
Scalable e-commerce web app
Scalable Episerver marketing website
Scalable Sitecore marketing website
Scalable Umbraco CMS web app
Scalable web apps with Redis
Simple branded website
Simple digital marketing website
Web and mobile apps with MySQL and Redis
Cloud Adoption Framework

Azure architecture icons

11/19/2021 • 2 minutes to read • Edit Online

Helping our customers design and architect new solutions is core to the Azure Architecture Center's mission. Architecture diagrams like those included in our guidance can help communicate design decisions and the relationships between components of a given workload. On this page you will find an official collection of Azure architecture icons including Azure product icons to help you build a custom architecture diagram for your next solution.

Do's

- Use the icon to illustrate how products can work together
- In diagrams, we recommend to include the product name somewhere close to the icon
- Use the icons as they would appear within Azure

Don'ts

- Don't crop, flip or rotate icons
- Don't distort or change icon shape in any way
- Don't use Microsoft product icons to represent your product or service

Example architecture diagram

Browse all Azure architectures to view other examples.

Icon updates

November 2020

The folder structure of our collection of Azure architecture icons has changed. The FAQs and Terms of Use PDF files appear in the first level when you download the SVG icons below. The files in the icons folder are the same except there is no longer a CXP folder. If you encounter any issues, let us know.

January 2021

There are ~26 icons that have been added to the existing set. The download file name has been updated to Azure_Public_Service_Icons_V4.zip

Icon terms

Microsoft permits the use of these icons in architectural diagrams, training materials, or documentation. You may copy, distribute, and display the icons only for the permitted use unless granted explicit permission by Microsoft. Microsoft reserves all other rights.

I agree to the above terms

DOWNLOAD SVG ICONS

More icon sets from Microsoft

- Microsoft 365 architecture icons and templates
- Dynamics 365 icons
- Microsoft Power Platform icons

What's new in Azure Architecture Center

11/19/2021 • 6 minutes to read • Edit Online

The Azure Architecture Center (AAC) helps you design, build, and operate solutions on Azure. Learn about the cloud architectural styles and design patterns. Use the technology choices and guides to decide the services that are right for your solution. The guidance is based on all aspects of building for the cloud, such as operations, security, reliability, performance, and cost optimization.

New and updated articles in Azure Architecture Center.

November 2021

New Articles

- Azure Kubernetes Service (AKS) and operational excellence
- Azure Kubernetes Service (AKS) and performance efficiency
- Automate IP rotation with Azure Container Instances container groups
- Azure Batch and operational excellence
- Azure Batch and performance efficiency
- Azure Batch and reliability
- Azure Kubernetes Service (AKS) and cost optimization
- Azure Kubernetes Service (AKS) and reliability
- Azure Kubernetes Service (AKS) and security
- Transactional Outbox pattern with Azure Cosmos DB
- Azure App Service and cost optimization
- Azure App Service and operational excellence
- Azure App Service and reliability
- Enterprise monitoring with Azure Monitor
- Scale AI and machine learning initiatives in regulated industries
- Long-term security log retention with Azure Data Explorer
- Multilayered protection for Azure virtual machines
- Automated guided vehicles fleet control
- High throughput stream ingestion to Azure Synapse
- Azure AD join for Azure Virtual Desktop
- Azure VMware Solution capacity planning
- Azure VMware Solution landing zone
- Azure VMware Solution networking

Updated Articles

- Data cache (#2df1e66db)
- Security alerts in Azure (#36456d4f5)
- AKS regulated cluster for PCI-DSS 3.2.1 - Data protection (#36456d4f5)
- AKS regulated cluster for PCI-DSS 3.2.1 - Network segmentation (#36456d4f5)
- Baseline architecture for an AKS cluster (#36456d4f5)
- Azure Functions security (#36456d4f5)

- Cost estimates for networking resources (#3970f3f69)
- Azure Well-Architected Framework review of Azure Application Gateway (#3970f3f69)
- Extract, transform, and load (ETL) using HDInsight (#c3735c005)
- Back up files and apps on Azure Stack Hub (#d89cee609)
- Instant broadcasting on serverless architecture (#b1545f035)
- Centralized app configuration and security (#1431f1f57)
- Use Azure HDInsight Hadoop Cluster on 1-TB dataset (#e302bb2fa)
- Parallel bulk data import in SQL partition tables (#e302bb2fa)
- Hybrid file services (#410dbde8d)
- Process Azure blob data with advanced analytics (#204f47047)
- Supply chain track and trace (#956a1f2d8)

October 2021

New Articles

- Client certificate for an Azure AD access token
- Apache NiFi on Azure
- Helm-based deployments for Apache NiFi
- Apache NiFi monitoring with MonitoFi
- Architecture for startups
- Customer lifetime value and churn prediction with Azure AI services
- Refactor mainframe applications with Advanced
- Transit hub dynamic pub-sub messaging system
- Mainframe and midrange data replication to Azure using Qlik
- AIX UNIX on-premises to Azure Linux migration
- Azure Storage considerations for multitenancy
- Predict hospital readmissions by using machine learning
- Power Automate deployment at scale
- Multi-tier app service with service endpoint
- Azure SQL Database considerations for multitenancy
- Efficient Docker image deployment for intermittent low-bandwidth connectivity
- Citizen AI with the Power Platform
- Stream processing with fully managed open-source data engines
- Event Hubs with Azure Functions
- Monitor Azure Functions and Event Hubs
- Performance and scale guidance for Event Hubs with Azure Functions
- Resilient design guidance for Event Hubs and Functions
- Secure Azure Functions with Event Hubs

Updated Articles

- Azure Machine Learning Anomaly Detection API (#3f89b0c77)
- Create personalized marketing solutions (#7f6cc7678)
- Deploy machine learning models to AKS with Kubeflow (#69221aa50)
- Predict loan charge-offs with HDInsight Spark (#3398fff02)
- Interactive voice response app with bot (#a4532949f)
- Demand forecasting and price optimization (#b2cd92b5d)
- Commerce chatbot for customer service (#ca079d0bf)

- Defender for Cloud Apps and Microsoft Sentinel for AWS (#15f56b812)
- Azure AD security for AWS (#e5279fc93)
- Administrative account security (#fcad2a7dc)
- Deploy AI-based footfall detection solution in Azure and Azure Stack Hub (#5c73ee7ee)
- Big data analytics with enterprise-grade security using Azure Synapse (#24ad8e000)
- Choose a batch processing technology (#8a7513ed1)
- Design for scaling (#f90bc6dea)
- Overview of the operational excellence pillar (#79ea619ac)

September 2021

New Articles

- Migrate master data services to Azure with CluedIn and Azure Purview
- Cost savings through HTAP with Azure SQL
- Azure Cosmos DB considerations for multitenancy
- Elastic Workplace Search on Azure
- JBoss deployment with Red Hat on Azure
- SAP workload automation using SUSE on Azure
- Create smart places by using Azure Digital Twins
- Secure research environment for regulated data
- Well-Architected Recommendation Process Guidance
- Principles of cost optimization
- Azure Well-Architected Framework review of an Azure NAT gateway
- Move an IoT solution from test to production
- Connected factory hierarchy service
- Overview of the reliability pillar
- Azure IoT client SDK support for third-party token servers
- Building blocks for autonomous-driving simulation environments
- CI/CD for Microsoft Power Platform
- Build a scalable system for massive data
- Minimal storage – change feed to replicate data
- Two-region web application with Table Storage failover
- Multi-region web application with Cosmos DB replication
- Multi-region web application with custom Storage Table replication
- Optimized storage with logical data classification
- Optimized storage – time based with Data Lake
- Optimized storage – time based - multi writes
- Azure Private Link in a hub-and-spoke network
- Tradeoffs for security
- Architectural approaches for storage and data in multitenant solutions

Updated Articles

- Zero-trust landing zone in Azure (#77cac625f)
- Governance, risk, and compliance (#77cac625f)
- Roles, responsibilities, and permissions (#77cac625f)
- Regulatory compliance (#77cac625f)
- Data protection in Azure (#77cac625f)

- Security audits (#77cac625f)
- Monitor Azure resources in Microsoft Defender for Cloud (#77cac625f)
- Security Operations Center (SOC or SecOps) monitoring in Azure (#77cac625f)
- Overview of the security pillar (#77cac625f)
- Transformer and collector ARM template (#44a45d12d)
- Overview of the cost optimization pillar (#c943d5252)
- Overview of a hybrid workload (#085dc36a7)
- Performance tuning - Multiple backend services (#0626db61b)
- Distributed business transaction performance tuning (#0626db61b)
- Key and secret management in Azure (#3a69bfa7b)
- Monitor application health for reliability (#bf16ced30)
- Alerting for DevOps (#40189ce5f)
- Performance efficiency checklist (#20dac17b3)
- Business Metrics (#762226372)
- Secure app configuration and dependencies (#291fdccd6)
- Compare AWS and Azure networking options (#1a8385188)
- Google Cloud to Azure services comparison (#917aca9a4)
- Solutions for the retail industry (#b605ffda8)
- Overview of the operational excellence pillar (#515eb84c7)
- Performance efficiency pillar overview (#515eb84c7)
- Azure security monitoring tools (#9b144df64)
- Data store decision tree (#c8c663fca)

August 2021

New Articles

- Zero-trust landing zone in Azure
- Security alerts in Azure
- Remediate security risks in Microsoft Defender for Cloud
- Monitor Azure resources in Microsoft Defender for Cloud
- Oracle Database with Azure NetApp Files
- SQL Server on Azure Virtual Machines with Azure NetApp Files
- Hybrid architecture design
- Data science and machine learning with Azure Databricks
- Orchestrate MLOps on Azure Databricks using Databricks Notebook
- Azure Well-Architected Framework review of Azure Firewall
- Re-engineer IBM z/OS batch applications on Azure
- Noisy Neighbor antipattern
- Getting started with Azure IoT solutions
- General mainframe refactor to Azure

Updated Articles

- Tradeoffs for performance efficiency (#ff65cd168)
- Secure app configuration and dependencies (#ff65cd168)
- Governance, risk, and compliance (#ff65cd168)
- Establish segmentation with management groups (#ff65cd168)
- Network security strategies on Azure (#ff65cd168)

- Regulatory compliance (#ff65cd168)
- Key and secret management in Azure (#ff65cd168)
- Security audits (#ff65cd168)
- Security Operations Center (SOC or SecOps) monitoring in Azure (#ff65cd168)
- Azure security test practices (#ff65cd168)
- Security monitoring in Azure (#ff65cd168)
- Authentication in multitenant applications (#8eac12b0f)
- Authorization in multitenant applications (#8eac12b0f)
- Sign-up and onboarding in a multi-tenant app (#8eac12b0f)
- Cache access tokens in a multitenant app (#8eac12b0f)
- Secure a backend web API in a multitenant app (#8eac12b0f)
- Performance testing and antipatterns (#96c15437c)
- Authentication with Azure AD (#8bc71d705)
- Azure mainframe and midrange architecture concepts and patterns (#76e02bca0)
- Services for securing network connectivity (#6b7484ad8)
- Azure control plane security (#8ae89d147)
- Code deployment security considerations in Azure (#af1679df0)
- Applications and services (#56885e80d)
- Data encryption in Azure (#ffe24f923)
- Traffic flow security in Azure (#856070254)
- Roles, responsibilities, and permissions (#732955401)
- Extend an on-premises network using VPN (#2bad97abf)
- Google Cloud to Azure services comparison (#7d2ca66d0)
- Application threat analysis (#982aeaa4c)
- Best practices for endpoint security (#fc2626dff)
- Governance considerations for secure deployment in Azure (#d5a4f981b)
- Multi-tier web application built for HA/DR (#250a84acc)
- Azure Kubernetes Service (AKS) design (#784eca7ef)
- High Performance Computing (HPC) on Azure (#fec1f53c4)
- AWS to Azure services comparison (#bac7c5914)
- Implement network segmentation patterns (#25ecd8538)
- Unisys Dorado mainframe migration to Azure with Astadia and Micro Focus (#c3d7d6875)
- Big compute architecture style (#e05914861)
- Serverless Functions reference architectures (#1ec2c2e0b)
- Deploy highly available Kubernetes cluster on Azure Stack Hub (#7b1fd4002)
- Related resources for multitenancy (#78c8b0fda)

Azure application architecture fundamentals

11/19/2021 • 3 minutes to read • Edit Online

This library of content presents a structured approach for designing applications on Azure that are scalable, secure, resilient, and highly available. The guidance is based on proven practices that we have learned from customer engagements.

Introduction

The cloud is changing how applications are designed and secured. Instead of monoliths, applications are decomposed into smaller, decentralized services. These services communicate through APIs or by using asynchronous messaging or eventing. Applications scale horizontally, adding new instances as demand requires.

These trends bring new challenges. Application states are distributed. Operations are done in parallel and asynchronously. Applications must be resilient when failures occur. Malicious actors continuously target applications. Deployments must be automated and predictable. Monitoring and telemetry are critical for gaining insight into the system. This guide is designed to help you navigate these changes.

TRADITIONAL ON-PREMISES	MODERN CLOUD
Monolithic	Decomposed
Designed for predictable scalability	Designed for elastic scale
Relational database	Polyglot persistence (mix of storage technologies)
Synchronized processing	Asynchronous processing
Design to avoid failures (MTBF)	Design for failure (MTTR)
Occasional large updates	Frequent small updates
Manual management	Automated self-management
Snowflake servers	Immutable infrastructure

How this guidance is structured

The Azure application architecture fundamentals guidance is organized as a series of steps, from the architecture and design to implementation. For each step, there is supporting guidance that will help you with the design of your application architecture.

Architecture styles

The first decision point is the most fundamental. What kind of architecture are you building? It might be a microservices architecture, a more traditional N-tier application, or a big data solution. We have identified several distinct architecture styles. There are benefits and challenges to each.

Learn more: Architecture styles

Technology choices

Knowing the type of architecture you are building, now you can start to choose the main technology pieces for the architecture. The following technology choices are critical:

- *Compute* refers to the hosting model for the computing resources that your applications run on. For more information, see Choose a compute service.
- *Data stores* include databases but also storage for message queues, caches, logs, and anything else that an application might persist to storage. For more information, see Choose a data store.
- *Messaging* technologies enable asynchronous messages between components of the system. For more information, see Choose a messaging service.

You will probably have to make additional technology choices along the way, but these three elements (compute, data, and messaging) are central to most cloud applications and will determine many aspects of your design.

Design the architecture

Once you have chosen the architecture style and the major technology components, you are ready to tackle the specific design of your application. Every application is different, but the following resources can help you along the way:

Reference architectures

Depending on your scenario, one of our reference architectures may be a good starting point. Each reference architecture includes recommended practices, along with considerations for scalability, availability, security, resilience, and other aspects of the design. Most also include a deployable solution or reference implementation.

Design principles

We have identified 10 high-level design principles that will make your application more scalable, resilient, and manageable. These design principles apply to any architecture style. Throughout the design process, keep these 10 high-level design principles in mind. For more information, see Design principles.

Design patterns

Software design patterns are repeatable patterns that are proven to solve specific problems. Our catalog of Cloud design patterns addresses specific challenges in distributed systems. They address aspects such as availability, high availability, operational excellence, resiliency, performance, and security. You can find our catalog of design patterns here.

Best practices

Our best practices articles cover various design considerations including API design, autoscaling, data partitioning, caching, and so forth. Review these and apply the best practices that are appropriate for your application.

Security best practices

Our security best practices describe how to ensure that the confidentiality, integrity, and availability of your application aren't compromised by malicious actors.

Quality pillars

A successful cloud application will focus on five pillars of software quality: Cost optimization, Operational excellence, Performance efficiency, Reliability, and Security.

Leverage the Microsoft Azure Well-Architected Framework to assess your architecture across these five pillars.

Next steps

Architecture styles

Architecture styles

11/19/2021 • 5 minutes to read • Edit Online

An *architecture style* is a family of architectures that share certain characteristics. For example, N-tier is a common architecture style. More recently, microservice architectures have started to gain favor. Architecture styles don't require the use of particular technologies, but some technologies are well-suited for certain architectures. For example, containers are a natural fit for microservices.

We have identified a set of architecture styles that are commonly found in cloud applications. The article for each style includes:

- A description and logical diagram of the style.
- Recommendations for when to choose this style.
- Benefits, challenges, and best practices.
- A recommended deployment using relevant Azure services.

A quick tour of the styles

This section gives a quick tour of the architecture styles that we've identified, along with some high-level considerations for their use. Read more details in the linked topics.

N-tier

N-tier is a traditional architecture for enterprise applications. Dependencies are managed by dividing the application into *layers* that perform logical functions, such as presentation, business logic, and data access. A layer can only call into layers that sit below it. However, this horizontal layering can be a liability. It can be hard to introduce changes in one part of the application without touching the rest of the application. That makes frequent updates a challenge, limiting how quickly new features can be added.

N-tier is a natural fit for migrating existing applications that already use a layered architecture. For that reason, N-tier is most often seen in infrastructure as a service (IaaS) solutions, or application that use a mix of IaaS and managed services.

Web-Queue-Worker

For a purely PaaS solution, consider a Web-Queue-Worker architecture. In this style, the application has a web front end that handles HTTP requests and a back-end worker that performs CPU-intensive tasks or long-running operations. The front end communicates to the worker through an asynchronous message queue.

Web-queue-worker is suitable for relatively simple domains with some resource-intensive tasks. Like N-tier, the architecture is easy to understand. The use of managed services simplifies deployment and operations. But with complex domains, it can be hard to manage dependencies. The front end and the worker can easily become large, monolithic components that are hard to maintain and update. As with N-tier, this can reduce the frequency of updates and limit innovation.

Microservices

If your application has a more complex domain, consider moving to a Microservices architecture. A microservices application is composed of many small, independent services. Each service implements a single business capability. Services are loosely coupled, communicating through API contracts.

Each service can be built by a small, focused development team. Individual services can be deployed without a lot of coordination between teams, which encourages frequent updates. A microservice architecture is more complex to build and manage than either N-tier or web-queue-worker. It requires a mature development and

DevOps culture. But done right, this style can lead to higher release velocity, faster innovation, and a more resilient architecture.

Event-driven architecture

Event-Driven Architectures use a publish-subscribe (pub-sub) model, where producers publish events, and consumers subscribe to them. The producers are independent from the consumers, and consumers are independent from each other.

Consider an event-driven architecture for applications that ingest and process a large volume of data with very low latency, such as IoT solutions. The style is also useful when different subsystems must perform different types of processing on the same event data.

Big Data, Big Compute

Big Data and Big Compute are specialized architecture styles for workloads that fit certain specific profiles. Big data divides a very large dataset into chunks, performing parallel processing across the entire set, for analysis and reporting. Big compute, also called high-performance computing (HPC), makes parallel computations across a large number (thousands) of cores. Domains include simulations, modeling, and 3-D rendering.

Architecture styles as constraints

An architecture style places constraints on the design, including the set of elements that can appear and the allowed relationships between those elements. Constraints guide the "shape" of an architecture by restricting the universe of choices. When an architecture conforms to the constraints of a particular style, certain desirable properties emerge.

For example, the constraints in microservices include:

- A service represents a single responsibility.
- Every service is independent of the others.
- Data is private to the service that owns it. Services do not share data.

By adhering to these constraints, what emerges is a system where services can be deployed independently, faults are isolated, frequent updates are possible, and it's easy to introduce new technologies into the application.

Before choosing an architecture style, make sure that you understand the underlying principles and constraints of that style. Otherwise, you can end up with a design that conforms to the style at a superficial level, but does not achieve the full potential of that style. It's also important to be pragmatic. Sometimes it's better to relax a constraint, rather than insist on architectural purity.

The following table summarizes how each style manages dependencies, and the types of domain that are best suited for each.

ARCHITECTURE STYLE	DEPENDENCY MANAGEMENT	DOMAIN TYPE
N-tier	Horizontal tiers divided by subnet	Traditional business domain. Frequency of updates is low.
Web-Queue-Worker	Front and backend jobs, decoupled by async messaging.	Relatively simple domain with some resource intensive tasks.
Microservices	Vertically (functionally) decomposed services that call each other through APIs.	Complicated domain. Frequent updates.

ARCHITECTURE STYLE	DEPENDENCY MANAGEMENT	DOMAIN TYPE
Event-driven architecture.	Producer/consumer. Independent view per sub-system.	IoT and real-time systems
Big data	Divide a huge dataset into small chunks. Parallel processing on local datasets.	Batch and real-time data analysis. Predictive analysis using ML.
Big compute	Data allocation to thousands of cores.	Compute intensive domains such as simulation.

Consider challenges and benefits

Constraints also create challenges, so it's important to understand the trade-offs when adopting any of these styles. Do the benefits of the architecture style outweigh the challenges, *for this subdomain and bounded context*.

Here are some of the types of challenges to consider when selecting an architecture style:

- **Complexity**. Is the complexity of the architecture justified for your domain? Conversely, is the style too simplistic for your domain? In that case, you risk ending up with a "big ball of mud", because the architecture does not help you to manage dependencies cleanly.
- **Asynchronous messaging and eventual consistency**. Asynchronous messaging can be used to decouple services, and increase reliability (because messages can be retried) and scalability. However, this also creates challenges in handling eventual consistency, as well as the possibility of duplicate messages.
- **Inter-service communication**. As you decompose an application into separate services, there is a risk that communication between services will cause unacceptable latency or create network congestion (for example, in a microservices architecture).
- **Manageability**. How hard is it to manage the application, monitor, deploy updates, and so on?

Big compute architecture style

11/19/2021 • 3 minutes to read • Edit Online

The term *big compute* describes large-scale workloads that require a large number of cores, often numbering in the hundreds or thousands. Scenarios include image rendering, fluid dynamics, financial risk modeling, oil exploration, drug design, and engineering stress analysis, among others.

Here are some typical characteristics of big compute applications:

- The work can be split into discrete tasks, which can be run across many cores simultaneously.
- Each task is finite. It takes some input, does some processing, and produces output. The entire application runs for a finite amount of time (minutes to days). A common pattern is to provision a large number of cores in a burst, and then spin down to zero once the application completes.
- The application does not need to stay up 24/7. However, the system must handle node failures or application crashes.
- For some applications, tasks are independent and can run in parallel. In other cases, tasks are tightly coupled, meaning they must interact or exchange intermediate results. In that case, consider using high-speed networking technologies such as InfiniBand and remote direct memory access (RDMA).
- Depending on your workload, you might use compute-intensive VM sizes (H16r, H16mr, and A9).

When to use this architecture

- Computationally intensive operations such as simulation and number crunching.
- Simulations that are computationally intensive and must be split across CPUs in multiple computers (10-1000s).
- Simulations that require too much memory for one computer, and must be split across multiple computers.
- Long-running computations that would take too long to complete on a single computer.
- Smaller computations that must be run 100s or 1000s of times, such as Monte Carlo simulations.

Benefits

- High performance with "embarrassingly parallel" processing.
- Can harness hundreds or thousands of computer cores to solve large problems faster.

- Access to specialized high-performance hardware, with dedicated high-speed InfiniBand networks.
- You can provision VMs as needed to do work, and then tear them down.

Challenges

- Managing the VM infrastructure.
- Managing the volume of number crunching
- Provisioning thousands of cores in a timely manner.
- For tightly coupled tasks, adding more cores can have diminishing returns. You may need to experiment to find the optimum number of cores.

Big compute using Azure Batch

Azure Batch is a managed service for running large-scale high-performance computing (HPC) applications.

Using Azure Batch, you configure a VM pool, and upload the applications and data files. Then the Batch service provisions the VMs, assign tasks to the VMs, runs the tasks, and monitors the progress. Batch can automatically scale out the VMs in response to the workload. Batch also provides job scheduling.

Big compute running on Virtual Machines

You can use Microsoft HPC Pack to administer a cluster of VMs, and schedule and monitor HPC jobs. With this approach, you must provision and manage the VMs and network infrastructure. Consider this approach if you have existing HPC workloads and want to move some or all it to Azure. You can move the entire HPC cluster to Azure, or you can keep your HPC cluster on-premises but use Azure for burst capacity. For more information, see Batch and HPC solutions for large-scale computing workloads.

HPC Pack deployed to Azure

In this scenario, the HPC cluster is created entirely within Azure.

The head node provides management and job scheduling services to the cluster. For tightly coupled tasks, use an RDMA network that provides very high bandwidth, low latency communication between VMs. For more information, see Deploy an HPC Pack 2016 cluster in Azure.

Burst an HPC cluster to Azure

In this scenario, an organization is running HPC Pack on-premises, and uses Azure VMs for burst capacity. The cluster head node is on-premises. ExpressRoute or VPN Gateway connects the on-premises network to the Azure VNet.

Next steps

- Choose an Azure compute service for your application
- High Performance Computing (HPC) on Azure
- HPC cluster deployed in the cloud

Big data architecture style

11/19/2021 • 10 minutes to read • Edit Online

A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems.

Big data solutions typically involve one or more of the following types of workload:

- Batch processing of big data sources at rest.
- Real-time processing of big data in motion.
- Interactive exploration of big data.
- Predictive analytics and machine learning.

Most big data architectures include some or all of the following components:

- **Data sources**: All big data solutions start with one or more data sources. Examples include:
  - Application data stores, such as relational databases.
  - Static files produced by applications, such as web server log files.
  - Real-time data sources, such as IoT devices.
- **Data storage**: Data for batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats. This kind of store is often called a *data lake*. Options for implementing this storage include Azure Data Lake Store or blob containers in Azure Storage.
- **Batch processing**: Because the data sets are so large, often a big data solution must process data files using long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. Usually these jobs involve reading source files, processing them, and writing the output to new files. Options include running U-SQL jobs in Azure Data Lake Analytics, using Hive, Pig, or custom Map/Reduce jobs in an HDInsight Hadoop cluster, or using Java, Scala, or Python programs in an HDInsight Spark cluster.
- **Real-time message ingestion**: If the solution includes real-time sources, the architecture must include a way to capture and store real-time messages for stream processing. This might be a simple data store, where incoming messages are dropped into a folder for processing. However, many solutions need a message ingestion store to act as a buffer for messages, and to support scale-out processing, reliable delivery, and other message queuing semantics. Options include Azure Event Hubs, Azure IoT Hubs, and Kafka.
- **Stream processing**: After capturing real-time messages, the solution must process them by filtering, aggregating, and otherwise preparing the data for analysis. The processed stream data is then written to an output sink. Azure Stream Analytics provides a managed stream processing service based on

perpetually running SQL queries that operate on unbounded streams. You can also use open source Apache streaming technologies like Storm and Spark Streaming in an HDInsight cluster.

- **Analytical data store**: Many big data solutions prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools. The analytical data store used to serve these queries can be a Kimball-style relational data warehouse, as seen in most traditional business intelligence (BI) solutions. Alternatively, the data could be presented through a low-latency NoSQL technology such as HBase, or an interactive Hive database that provides a metadata abstraction over data files in the distributed data store. Azure Synapse Analytics provides a managed service for large-scale, cloud-based data warehousing. HDInsight supports Interactive Hive, HBase, and Spark SQL, which can also be used to serve data for analysis.

- **Analysis and reporting**: The goal of most big data solutions is to provide insights into the data through analysis and reporting. To empower users to analyze the data, the architecture may include a data modeling layer, such as a multidimensional OLAP cube or tabular data model in Azure Analysis Services. It might also support self-service BI, using the modeling and visualization technologies in Microsoft Power BI or Microsoft Excel. Analysis and reporting can also take the form of interactive data exploration by data scientists or data analysts. For these scenarios, many Azure services support analytical notebooks, such as Jupyter, enabling these users to leverage their existing skills with Python or R. For large-scale data exploration, you can use Microsoft R Server, either standalone or with Spark.

- **Orchestration**: Most big data solutions consist of repeated data processing operations, encapsulated in workflows, that transform source data, move data between multiple sources and sinks, load the processed data into an analytical data store, or push the results straight to a report or dashboard. To automate these workflows, you can use an orchestration technology such Azure Data Factory or Apache Oozie and Sqoop.

Azure includes many services that can be used in a big data architecture. They fall roughly into two categories:

- Managed services, including Azure Data Lake Store, Azure Data Lake Analytics, Azure Synapse Analytics, Azure Stream Analytics, Azure Event Hub, Azure IoT Hub, and Azure Data Factory.
- Open source technologies based on the Apache Hadoop platform, including HDFS, HBase, Hive, Pig, Spark, Storm, Oozie, Sqoop, and Kafka. These technologies are available on Azure in the Azure HDInsight service.

These options are not mutually exclusive, and many solutions combine open source technologies with Azure services.

When to use this architecture

Consider this architecture style when you need to:

- Store and process data in volumes too large for a traditional database.
- Transform unstructured data for analysis and reporting.
- Capture, process, and analyze unbounded streams of data in real time, or with low latency.
- Use Azure Machine Learning or Microsoft Cognitive Services.

Benefits

- **Technology choices**. You can mix and match Azure managed services and Apache technologies in HDInsight clusters, to capitalize on existing skills or technology investments.
- **Performance through parallelism**. Big data solutions take advantage of parallelism, enabling high-performance solutions that scale to large volumes of data.
- **Elastic scale**. All of the components in the big data architecture support scale-out provisioning, so that you can adjust your solution to small or large workloads, and pay only for the resources that you use.

- **Interoperability with existing solutions**. The components of the big data architecture are also used for IoT processing and enterprise BI solutions, enabling you to create an integrated solution across data workloads.

Challenges

- **Complexity**. Big data solutions can be extremely complex, with numerous components to handle data ingestion from multiple data sources. It can be challenging to build, test, and troubleshoot big data processes. Moreover, there may be a large number of configuration settings across multiple systems that must be used in order to optimize performance.
- **Skillset**. Many big data technologies are highly specialized, and use frameworks and languages that are not typical of more general application architectures. On the other hand, big data technologies are evolving new APIs that build on more established languages. For example, the U-SQL language in Azure Data Lake Analytics is based on a combination of Transact-SQL and C#. Similarly, SQL-based APIs are available for Hive, HBase, and Spark.
- **Technology maturity**. Many of the technologies used in big data are evolving. While core Hadoop technologies such as Hive and Pig have stabilized, emerging technologies such as Spark introduce extensive changes and enhancements with each new release. Managed services such as Azure Data Lake Analytics and Azure Data Factory are relatively young, compared with other Azure services, and will likely evolve over time.
- **Security**. Big data solutions usually rely on storing all static data in a centralized data lake. Securing access to this data can be challenging, especially when the data must be ingested and consumed by multiple applications and platforms.

Best practices

- **Leverage parallelism**. Most big data processing technologies distribute the workload across multiple processing units. This requires that static data files are created and stored in a splittable format. Distributed file systems such as HDFS can optimize read and write performance, and the actual processing is performed by multiple cluster nodes in parallel, which reduces overall job times.

- **Partition data**. Batch processing usually happens on a recurring schedule — for example, weekly or monthly. Partition data files, and data structures such as tables, based on temporal periods that match the processing schedule. That simplifies data ingestion and job scheduling, and makes it easier to troubleshoot failures. Also, partitioning tables that are used in Hive, U-SQL, or SQL queries can significantly improve query performance.

- **Apply schema-on-read semantics**. Using a data lake lets you to combine storage for files in multiple formats, whether structured, semi-structured, or unstructured. Use *schema-on-read* semantics, which project a schema onto the data when the data is processing, not when the data is stored. This builds flexibility into the solution, and prevents bottlenecks during data ingestion caused by data validation and type checking.

- **Process data in-place**. Traditional BI solutions often use an extract, transform, and load (ETL) process to move data into a data warehouse. With larger volumes data, and a greater variety of formats, big data solutions generally use variations of ETL, such as transform, extract, and load (TEL). With this approach, the data is processed within the distributed data store, transforming it to the required structure, before moving the transformed data into an analytical data store.

- **Balance utilization and time costs**. For batch processing jobs, it's important to consider two factors: The per-unit cost of the compute nodes, and the per-minute cost of using those nodes to complete the job. For example, a batch job may take eight hours with four cluster nodes. However, it might turn out that the job uses all four nodes only during the first two hours, and after that, only two nodes are required. In that case, running the entire job on two nodes would increase the total job time, but would not double it, so the total cost would be less. In some business scenarios, a longer processing time may

be preferable to the higher cost of using underutilized cluster resources.

- **Separate cluster resources**. When deploying HDInsight clusters, you will normally achieve better performance by provisioning separate cluster resources for each type of workload. For example, although Spark clusters include Hive, if you need to perform extensive processing with both Hive and Spark, you should consider deploying separate dedicated Spark and Hadoop clusters. Similarly, if you are using HBase and Storm for low latency stream processing and Hive for batch processing, consider separate clusters for Storm, HBase, and Hadoop.

- **Orchestrate data ingestion**. In some cases, existing business applications may write data files for batch processing directly into Azure storage blob containers, where they can be consumed by HDInsight or Azure Data Lake Analytics. However, you will often need to orchestrate the ingestion of data from on-premises or external data sources into the data lake. Use an orchestration workflow or pipeline, such as those supported by Azure Data Factory or Oozie, to achieve this in a predictable and centrally manageable fashion.

- **Scrub sensitive data early**. The data ingestion workflow should scrub sensitive data early in the process, to avoid storing it in the data lake.

IoT architecture

Internet of Things (IoT) is a specialized subset of big data solutions. The following diagram shows a possible logical architecture for IoT. The diagram emphasizes the event-streaming components of the architecture.

The **cloud gateway** ingests device events at the cloud boundary, using a reliable, low latency messaging system.

Devices might send events directly to the cloud gateway, or through a **field gateway**. A field gateway is a specialized device or software, usually colocated with the devices, that receives events and forwards them to the cloud gateway. The field gateway might also preprocess the raw device events, performing functions such as filtering, aggregation, or protocol transformation.

After ingestion, events go through one or more **stream processors** that can route the data (for example, to storage) or perform analytics and other processing.

The following are some common types of processing. (This list is certainly not exhaustive.)

- Writing event data to cold storage, for archiving or batch analytics.

- Hot path analytics, analyzing the event stream in (near) real time, to detect anomalies, recognize patterns over rolling time windows, or trigger alerts when a specific condition occurs in the stream.

- Handling special types of non-telemetry messages from devices, such as notifications and alarms.

- Machine learning.

The boxes that are shaded gray show components of an IoT system that are not directly related to event streaming, but are included here for completeness.

- The **device registry** is a database of the provisioned devices, including the device IDs and usually device metadata, such as location.
- The **provisioning API** is a common external interface for provisioning and registering new devices.
- Some IoT solutions allow **command and control messages** to be sent to devices.

> This section has presented a very high-level view of IoT, and there are many subtleties and challenges to consider. For a more detailed reference architecture and discussion, see the Microsoft Azure IoT Reference Architecture (PDF download).

Next steps

- Learn more about big data architectures.
- Learn more about IoT solutions.

Event-driven architecture style

11/19/2021 • 3 minutes to read • Edit Online

An event-driven architecture consists of **event producers** that generate a stream of events, and **event consumers** that listen for the events.

Events are delivered in near real time, so consumers can respond immediately to events as they occur. Producers are decoupled from consumers — a producer doesn't know which consumers are listening. Consumers are also decoupled from each other, and every consumer sees all of the events. This differs from a Competing Consumers pattern, where consumers pull messages from a queue and a message is processed just once (assuming no errors). In some systems, such as IoT, events must be ingested at very high volumes.

An event driven architecture can use a pub/sub model or an event stream model.

- **Pub/sub**: The messaging infrastructure keeps track of subscriptions. When an event is published, it sends the event to each subscriber. After an event is received, it cannot be replayed, and new subscribers do not see the event.
- **Event streaming**: Events are written to a log. Events are strictly ordered (within a partition) and durable. Clients don't subscribe to the stream, instead a client can read from any part of the stream. The client is responsible for advancing its position in the stream. That means a client can join at any time, and can replay events.

On the consumer side, there are some common variations:

- **Simple event processing**. An event immediately triggers an action in the consumer. For example, you could use Azure Functions with a Service Bus trigger, so that a function executes whenever a message is published to a Service Bus topic.
- **Complex event processing**. A consumer processes a series of events, looking for patterns in the event data, using a technology such as Azure Stream Analytics or Apache Storm. For example, you could aggregate readings from an embedded device over a time window, and generate a notification if the moving average crosses a certain threshold.
- **Event stream processing**. Use a data streaming platform, such as Azure IoT Hub or Apache Kafka, as a pipeline to ingest events and feed them to stream processors. The stream processors act to process or transform the stream. There may be multiple stream processors for different subsystems of the application. This approach is a good fit for IoT workloads.

The source of the events may be external to the system, such as physical devices in an IoT solution. In that case, the system must be able to ingest the data at the volume and throughput that is required by the data source.

In the logical diagram above, each type of consumer is shown as a single box. In practice, it's common to have

multiple instances of a consumer, to avoid having the consumer become a single point of failure in system. Multiple instances might also be necessary to handle the volume and frequency of events. Also, a single consumer might process events on multiple threads. This can create challenges if events must be processed in order or require exactly-once semantics. See Minimize Coordination.

When to use this architecture

- Multiple subsystems must process the same events.
- Real-time processing with minimum time lag.
- Complex event processing, such as pattern matching or aggregation over time windows.
- High volume and high velocity of data, such as IoT.

Benefits

- Producers and consumers are decoupled.
- No point-to-point integrations. It's easy to add new consumers to the system.
- Consumers can respond to events immediately as they arrive.
- Highly scalable and distributed.
- Subsystems have independent views of the event stream.

Challenges

- Guaranteed delivery. In some systems, especially in IoT scenarios, it's crucial to guarantee that events are delivered.
- Processing events in order or exactly once. Each consumer type typically runs in multiple instances, for resiliency and scalability. This can create a challenge if the events must be processed in order (within a consumer type), or if the processing logic is not idempotent.

Additional considerations

- The amount of data to include in an event can be a significant consideration that affects both performance and cost. Putting all the relevant information needed for processing in the event itself can simplify the processing code and save additional lookups. Putting the minimal amount of information in an event, like just a couple of identifiers, will reduce transport time and cost, but requires the processing code to look up any additional information it needs. For more information on this, take a look at this blog post.

Microservices architecture style

11/19/2021 • 6 minutes to read • Edit Online

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.

What are microservices?

- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.

Besides for the services themselves, some other components appear in a typical microservices architecture:

Management/orchestration. This component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.

API Gateway. The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

Advantages of using an API gateway include:

- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.

- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.
- Out-of-the-box policies, like for throttling, caching, transformation, or validation.

Benefits

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features may be held up waiting for a bug fix to be integrated, tested, and published.
- **Small, focused teams**. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- **Small code base**. In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.
- **Mix of technologies**. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Fault isolation**. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).
- **Scalability**. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- **Data isolation**. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- **Complexity**. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing**. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- **Lack of governance**. The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place,

without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

- **Network congestion and latency**. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load leveling.
- **Data integrity**. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- **Management**. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
- **Versioning**. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
- **Skill set**. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

Best practices

- Model services around the business domain.
- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.
- Data storage should be private to the service that owns the data. Use the best storage for each service and data type.
- Services communicate through well-designed APIs. Avoid leaking implementation details. APIs should model the domain, not the internal implementation of the service.
- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.
- Offload cross-cutting concerns, such as authentication and SSL termination, to the gateway.
- Keep domain knowledge out of the gateway. The gateway should handle and route client requests without any knowledge of the business rules or domain logic. Otherwise, the gateway becomes a dependency and can cause coupling between services.
- Services should have loose coupling and high functional cohesion. Functions that are likely to change together should be packaged and deployed together. If they reside in separate services, those services end up being tightly coupled, because a change in one service will require updating the other service. Overly chatty communication between two services may be a symptom of tight coupling and low cohesion.
- Isolate failures. Use resiliency strategies to prevent failures within a service from cascading. See Resiliency patterns and Designing reliable applications.

Next steps

For detailed guidance about building a microservices architecture on Azure, see Designing, building, and operating microservices on Azure.

N-tier architecture style

11/19/2021 • 5 minutes to read • Edit Online

An N-tier architecture divides an application into **logical layers** and **physical tiers**.

Layers are a way to separate responsibilities and manage dependencies. Each layer has a specific responsibility. A higher layer can use services in a lower layer, but not the other way around.

Tiers are physically separated, running on separate machines. A tier can call to another tier directly, or use asynchronous messaging (message queue). Although each layer might be hosted in its own tier, that's not required. Several layers might be hosted on the same tier. Physically separating the tiers improves scalability and resiliency, but also adds latency from the additional network communication.

A traditional three-tier application has a presentation tier, a middle tier, and a database tier. The middle tier is optional. More complex applications can have more than three tiers. The diagram above shows an application with two middle tiers, encapsulating different areas of functionality.

An N-tier application can have a **closed layer architecture** or an **open layer architecture**:

- In a closed layer architecture, a layer can only call the next layer immediately down.
- In an open layer architecture, a layer can call any of the layers below it.

A closed layer architecture limits the dependencies between layers. However, it might create unnecessary network traffic, if one layer simply passes requests along to the next layer.

When to use this architecture

N-tier architectures are typically implemented as infrastructure-as-service (IaaS) applications, with each tier running on a separate set of VMs. However, an N-tier application doesn't need to be pure IaaS. Often, it's advantageous to use managed services for some parts of the architecture, particularly caching, messaging, and data storage.

Consider an N-tier architecture for:

- Simple web applications.
- Migrating an on-premises application to Azure with minimal refactoring.
- Unified development of on-premises and cloud applications.

N-tier architectures are very common in traditional on-premises applications, so it's a natural fit for migrating existing workloads to Azure.

Benefits

- Portability between cloud and on-premises, and between cloud platforms.
- Less learning curve for most developers.
- Natural evolution from the traditional application model.
- Open to heterogeneous environment (Windows/Linux)

Challenges

- It's easy to end up with a middle tier that just does CRUD operations on the database, adding extra latency without doing any useful work.
- Monolithic design prevents independent deployment of features.
- Managing an IaaS application is more work than an application that uses only managed services.
- It can be difficult to manage network security in a large system.

Best practices

- Use autoscaling to handle changes in load. See Autoscaling best practices.
- Use asynchronous messaging to decouple tiers.
- Cache semistatic data. See Caching best practices.
- Configure the database tier for high availability, using a solution such as SQL Server Always On availability groups.
- Place a web application firewall (WAF) between the front end and the Internet.
- Place each tier in its own subnet, and use subnets as a security boundary.
- Restrict access to the data tier, by allowing requests only from the middle tier(s).

N-tier architecture on virtual machines

This section describes a recommended N-tier architecture running on VMs.

Each tier consists of two or more VMs, placed in an availability set or virtual machine scale set. Multiple VMs provide resiliency in case one VM fails. Load balancers are used to distribute requests across the VMs in a tier. A tier can be scaled horizontally by adding more VMs to the pool.

Each tier is also placed inside its own subnet, meaning their internal IP addresses fall within the same address range. That makes it easy to apply network security group rules and route tables to individual tiers.

The web and business tiers are stateless. Any VM can handle any request for that tier. The data tier should consist of a replicated database. For Windows, we recommend SQL Server, using Always On availability groups for high availability. For Linux, choose a database that supports replication, such as Apache Cassandra.

Network security groups restrict access to each tier. For example, the database tier only allows access from the business tier.

> **NOTE**
>
> The layer labeled "Business Tier" in our reference diagram is a moniker to the business logic tier. Likewise, we also call the presentation tier the "Web Tier." In our example, this is a web application, though multi-tier architectures can be used for other topologies as well (like desktop apps). Name your tiers what works best for your team to communicate the intent of that logical and/or physical tier in your application - you could even express that naming in resources you choose to represent that tier (e.g. vmss-appName-business-layer).

For more information about running N-tier applications on Azure:

- Run Windows VMs for an N-tier application
- Windows N-tier application on Azure with SQL Server
- Microsoft Learn module: Tour the N-tier architecture style
- Azure Bastion

Additional considerations

- N-tier architectures are not restricted to three tiers. For more complex applications, it is common to have more tiers. In that case, consider using layer-7 routing to route requests to a particular tier.
- Tiers are the boundary of scalability, reliability, and security. Consider having separate tiers for services with different requirements in those areas.
- Use virtual machine scale sets for autoscaling.
- Look for places in the architecture where you can use a managed service without significant refactoring. In particular, look at caching, messaging, storage, and databases.
- For higher security, place a network DMZ in front of the application. The DMZ includes network virtual appliances (NVAs) that implement security functionality such as firewalls and packet inspection. For more information, see Network DMZ reference architecture.
- For high availability, place two or more NVAs in an availability set, with an external load balancer to distribute Internet requests across the instances. For more information, see Deploy highly available network virtual appliances.
- Do not allow direct RDP or SSH access to VMs that are running application code. Instead, operators should log into a jumpbox, also called a bastion host. This is a VM on the network that administrators use to connect to the other VMs. The jumpbox has a network security group that allows RDP or SSH only from approved public IP addresses.
- You can extend the Azure virtual network to your on-premises network using a site-to-site virtual private network (VPN) or Azure ExpressRoute. For more information, see Hybrid network reference architecture.
- If your organization uses Active Directory to manage identity, you may want to extend your Active Directory environment to the Azure VNet. For more information, see Identity management reference architecture.
- If you need higher availability than the Azure SLA for VMs provides, replicate the application across two regions and use Azure Traffic Manager for failover. For more information, see Run Windows VMs in multiple regions or Run Linux VMs in multiple regions.

Web-Queue-Worker architecture style

11/19/2021 • 3 minutes to read • Edit Online

The core components of this architecture are a **web front end** that serves client requests, and a **worker** that performs resource-intensive tasks, long-running workflows, or batch jobs. The web front end communicates with the worker through a **message queue**.

Other components that are commonly incorporated into this architecture include:

- One or more databases.
- A cache to store values from the database for quick reads.
- CDN to serve static content
- Remote services, such as email or SMS service. Often these are provided by third parties.
- Identity provider for authentication.

The web and worker are both stateless. Session state can be stored in a distributed cache. Any long-running work is done asynchronously by the worker. The worker can be triggered by messages on the queue, or run on a schedule for batch processing. The worker is an optional component. If there are no long-running operations, the worker can be omitted.

The front end might consist of a web API. On the client side, the web API can be consumed by a single-page application that makes AJAX calls, or by a native client application.

When to use this architecture

The Web-Queue-Worker architecture is typically implemented using managed compute services, either Azure App Service or Azure Cloud Services.

Consider this architecture style for:

- Applications with a relatively simple domain.
- Applications with some long-running workflows or batch operations.
- When you want to use managed services, rather than infrastructure as a service (IaaS).

Benefits

- Relatively simple architecture that is easy to understand.

- Easy to deploy and manage.
- Clear separation of concerns.
- The front end is decoupled from the worker using asynchronous messaging.
- The front end and the worker can be scaled independently.

Challenges

- Without careful design, the front end and the worker can become large, monolithic components that are difficult to maintain and update.
- There may be hidden dependencies, if the front end and worker share data schemas or code modules.

Best practices

- Expose a well-designed API to the client. See API design best practices.
- Autoscale to handle changes in load. See Autoscaling best practices.
- Cache semi-static data. See Caching best practices.
- Use a CDN to host static content. See CDN best practices.
- Use polyglot persistence when appropriate. See Use the best data store for the job.
- Partition data to improve scalability, reduce contention, and optimize performance. See Data partitioning best practices.

Web-Queue-Worker on Azure App Service

This section describes a recommended Web-Queue-Worker architecture that uses Azure App Service.

- The front end is implemented as an Azure App Service web app, and the worker is implemented as an Azure Functions app. The web app and the function app are both associated with an App Service plan that provides the VM instances.
- You can use either Azure Service Bus or Azure Storage queues for the message queue. (The diagram shows an Azure Storage queue.)
- Azure Cache for Redis stores session state and other data that needs low latency access.
- Azure CDN is used to cache static content such as images, CSS, or HTML.
- For storage, choose the storage technologies that best fit the needs of the application. You might use multiple storage technologies (polyglot persistence). To illustrate this idea, the diagram shows Azure SQL Database and Azure Cosmos DB.

For more details, see App Service web application reference architecture.

Additional considerations

- Not every transaction has to go through the queue and worker to storage. The web front end can perform simple read/write operations directly. Workers are designed for resource-intensive tasks or long-running workflows. In some cases, you might not need a worker at all.
- Use the built-in autoscale feature of App Service to scale out the number of VM instances. If the load on the application follows predictable patterns, use schedule-based autoscale. If the load is unpredictable, use metrics-based autoscaling rules.
- Consider putting the web app and the function app into separate App Service plans. That way, they can be scaled independently.
- Use separate App Service plans for production and testing. Otherwise, if you use the same plan for production and testing, it means your tests are running on your production VMs.
- Use deployment slots to manage deployments. This lets you to deploy an updated version to a staging slot, then swap over to the new version. It also lets you swap back to the previous version, if there was a problem with the update.

Ten design principles for Azure applications

11/19/2021 • 2 minutes to read • Edit Online

Follow these design principles to make your application more scalable, resilient, and manageable.

Design for self healing. In a distributed system, failures happen. Design your application to be self healing when failures occur.

Make all things redundant. Build redundancy into your application, to avoid having single points of failure.

Minimize coordination. Minimize coordination between application services to achieve scalability.

Design to scale out. Design your application so that it can scale horizontally, adding or removing new instances as demand requires.

Partition around limits. Use partitioning to work around database, network, and compute limits.

Design for operations. Design your application so that the operations team has the tools they need.

Use managed services. When possible, use platform as a service (PaaS) rather than infrastructure as a service (IaaS).

Use the best data store for the job. Pick the storage technology that is the best fit for your data and how it will be used.

Design for evolution. All successful applications change over time. An evolutionary design is key for continuous innovation.

Build for the needs of business. Every design decision must be justified by a business requirement.

Design for self healing

11/19/2021 • 4 minutes to read • Edit Online

Design your application to be self healing when failures occur

In a distributed system, failures can happen. Hardware can fail. The network can have transient failures. Rarely, an entire service or region may experience a disruption, but even those must be planned for.

Therefore, design an application to be self healing when failures occur. This requires a three-pronged approach:

- Detect failures.
- Respond to failures gracefully.
- Log and monitor failures, to give operational insight.

How you respond to a particular type of failure may depend on your application's availability requirements. For example, if you require very high availability, you might automatically fail over to a secondary region during a regional outage. However, that will incur a higher cost than a single-region deployment.

Also, don't just consider big events like regional outages, which are generally rare. You should focus as much, if not more, on handling local, short-lived failures, such as network connectivity failures or failed database connections.

Recommendations

Retry failed operations. Transient failures may occur due to momentary loss of network connectivity, a dropped database connection, or a timeout when a service is busy. Build retry logic into your application to handle transient failures. For many Azure services, the client SDK implements automatic retries. For more information, see Transient fault handling and the Retry pattern.

Protect failing remote services (Circuit Breaker). It's good to retry after a transient failure, but if the failure persists, you can end up with too many callers hammering a failing service. This can lead to cascading failures, as requests back up. Use the Circuit Breaker pattern to fail fast (without making the remote call) when an operation is likely to fail.

Isolate critical resources (Bulkhead). Failures in one subsystem can sometimes cascade. This can happen if a failure causes some resources, such as threads or sockets, not to get freed in a timely manner, leading to resource exhaustion. To avoid this, partition a system into isolated groups, so that a failure in one partition does not bring down the entire system.

Perform load leveling. Applications may experience sudden spikes in traffic that can overwhelm services on the backend. To avoid this, use the Queue-Based Load Leveling pattern to queue work items to run asynchronously. The queue acts as a buffer that smooths out peaks in the load.

Fail over. If an instance can't be reached, fail over to another instance. For things that are stateless, like a web server, put several instances behind a load balancer or traffic manager. For things that store state, like a database, use replicas and fail over. Depending on the data store and how it replicates, this may require the application to deal with eventual consistency.

Compensate failed transactions. In general, avoid distributed transactions, as they require coordination across services and resources. Instead, compose an operation from smaller individual transactions. If the operation fails midway through, use Compensating Transactions to undo any step that already completed.

Checkpoint long-running transactions. Checkpoints can provide resiliency if a long-running operation fails.

When the operation restarts (for example, it is picked up by another VM), it can be resumed from the last checkpoint.

Degrade gracefully. Sometimes you can't work around a problem, but you can provide reduced functionality that is still useful. Consider an application that shows a catalog of books. If the application can't retrieve the thumbnail image for the cover, it might show a placeholder image. Entire subsystems might be noncritical for the application. For example, in an e-commerce site, showing product recommendations is probably less critical than processing orders.

Throttle clients. Sometimes a small number of users create excessive load, which can reduce your application's availability for other users. In this situation, throttle the client for a certain period of time. See the Throttling pattern.

Block bad actors. Just because you throttle a client, it doesn't mean client was acting maliciously. It just means the client exceeded their service quota. But if a client consistently exceeds their quota or otherwise behaves badly, you might block them. Define an out-of-band process for user to request getting unblocked.

Use leader election. When you need to coordinate a task, use Leader Election to select a coordinator. That way, the coordinator is not a single point of failure. If the coordinator fails, a new one is selected. Rather than implement a leader election algorithm from scratch, consider an off-the-shelf solution such as Zookeeper.

Test with fault injection. All too often, the success path is well tested but not the failure path. A system could run in production for a long time before a failure path is exercised. Use fault injection to test the resiliency of the system to failures, either by triggering actual failures or by simulating them.

Embrace chaos engineering. Chaos engineering extends the notion of fault injection, by randomly injecting failures or abnormal conditions into production instances.

For a structured approach to making your applications self healing, see Design reliable applications for Azure.

Make all things redundant

11/19/2021 • 2 minutes to read • Edit Online

Build redundancy into your application, to avoid having single points of failure

A resilient application routes around failure. Identify the critical paths in your application. Is there redundancy at each point in the path? When a subsystem fails, will the application fail over to something else?

Recommendations

Consider business requirements. The amount of redundancy built into a system can affect both cost and complexity. Your architecture should be informed by your business requirements, such as recovery time objective (RTO). For example, a multi-region deployment is more expensive than a single-region deployment, and is more complicated to manage. You will need operational procedures to handle failover and failback. The additional cost and complexity might be justified for some business scenarios and not others.

Place VMs behind a load balancer. Don't use a single VM for mission-critical workloads. Instead, place multiple VMs behind a load balancer. If any VM becomes unavailable, the load balancer distributes traffic to the remaining healthy VMs. To learn how to deploy this configuration, see Multiple VMs for scalability and availability.

Replicate databases. Azure SQL Database and Cosmos DB automatically replicate the data within a region, and you can enable geo-replication across regions. If you are using an IaaS database solution, choose one that supports replication and failover, such as SQL Server Always On availability groups.

Enable geo-replication. Geo-replication for Azure SQL Database and Cosmos DB creates secondary readable replicas of your data in one or more secondary regions. In the event of an outage, the database can fail over to the secondary region for writes.

Partition for availability. Database partitioning is often used to improve scalability, but it can also improve availability. If one shard goes down, the other shards can still be reached. A failure in one shard will only disrupt a subset of the total transactions.

Deploy to more than one region. For the highest availability, deploy the application to more than one region. That way, in the rare case when a problem affects an entire region, the application can fail over to another region. The following diagram shows a multi-region application that uses Azure Traffic Manager to handle failover.

Synchronize front and backend failover. Use Azure Traffic Manager to fail over the front end. If the front end becomes unreachable in one region, Traffic Manager will route new requests to the secondary region. Depending on your database solution, you may need to coordinate failing over the database.

Use automatic failover but manual failback. Use Traffic Manager for automatic failover, but not for automatic failback. Automatic failback carries a risk that you might switch to the primary region before the region is completely healthy. Instead, verify that all application subsystems are healthy before manually failing back. Also, depending on the database, you might need to check data consistency before failing back.

Include redundancy for Traffic Manager. Traffic Manager is a possible failure point. Review the Traffic Manager SLA, and determine whether using Traffic Manager alone meets your business requirements for high availability. If not, consider adding another traffic management solution as a failback. If the Azure Traffic Manager service fails, change your CNAME records in DNS to point to the other traffic management service.

Minimize coordination

11/19/2021 • 4 minutes to read • Edit Online

Minimize coordination between application services to achieve scalability

Most cloud applications consist of multiple application services — web front ends, databases, business processes, reporting and analysis, and so on. To achieve scalability and reliability, each of those services should run on multiple instances.

What happens when two instances try to perform concurrent operations that affect some shared state? In some cases, there must be coordination across nodes, for example to preserve ACID guarantees. In this diagram, `Node2` is waiting for `Node1` to release a database lock:

Coordination limits the benefits of horizontal scale and creates bottlenecks. In this example, as you scale out the application and add more instances, you'll see increased lock contention. In the worst case, the front-end instances will spend most of their time waiting on locks.

"Exactly once" semantics are another frequent source of coordination. For example, an order must be processed exactly once. Two workers are listening for new orders. `Worker1` picks up an order for processing. The application must ensure that `Worker2` doesn't duplicate the work, but also if `Worker1` crashes, the order isn't dropped.

You can use a pattern such as Scheduler Agent Supervisor to coordinate between the workers, but in this case a better approach might be to partition the work. Each worker is assigned a certain range of orders (say, by billing region). If a worker crashes, a new instance picks up where the previous instance left off, but multiple instances aren't contending.

Recommendations

Embrace eventual consistency. When data is distributed, it takes coordination to enforce strong consistency guarantees. For example, suppose an operation updates two databases. Instead of putting it into a single transaction scope, it's better if the system can accommodate eventual consistency, perhaps by using the

Compensating Transaction pattern to logically roll back after a failure.

Use domain events to synchronize state. A domain event is an event that records when something happens that has significance within the domain. Interested services can listen for the event, rather than using a global transaction to coordinate across multiple services. If this approach is used, the system must tolerate eventual consistency (see previous item).

Consider patterns such as CQRS and event sourcing. These two patterns can help to reduce contention between read workloads and write workloads.

- The CQRS pattern separates read operations from write operations. In some implementations, the read data is physically separated from the write data.
- In the Event Sourcing pattern, state changes are recorded as a series of events to an append-only data store. Appending an event to the stream is an atomic operation, requiring minimal locking.

These two patterns complement each other. If the write-only store in CQRS uses event sourcing, the read-only store can listen for the same events to create a readable snapshot of the current state, optimized for queries. Before adopting CQRS or event sourcing, however, be aware of the challenges of this approach.

Partition data. Avoid putting all of your data into one data schema that is shared across many application services. A microservices architecture enforces this principle by making each service responsible for its own data store. Within a single database, partitioning the data into shards can improve concurrency, because a service writing to one shard does not affect a service writing to a different shard.

Design idempotent operations. When possible, design operations to be idempotent. That way, they can be handled using at-least-once semantics. For example, you can put work items on a queue. If a worker crashes in the middle of an operation, another worker simply picks up the work item.

Use asynchronous parallel processing. If an operation requires multiple steps that are performed asynchronously (such as remote service calls), you might be able to call them in parallel, and then aggregate the results. This approach assumes that each step does not depend on the results of the previous step.

Use optimistic concurrency when possible. Pessimistic concurrency control uses database locks to prevent conflicts. This can cause poor performance and reduce availability. With optimistic concurrency control, each transaction modifies a copy or snapshot of the data. When the transaction is committed, the database engine validates the transaction and rejects any transactions that would affect database consistency.

Azure SQL Database and SQL Server support optimistic concurrency through snapshot isolation. Some Azure storage services support optimistic concurrency through the use of Etags, including Azure Cosmos DB and Azure Storage.

Consider MapReduce or other parallel, distributed algorithms. Depending on the data and type of work to be performed, you may be able to split the work into independent tasks that can be performed by multiple nodes working in parallel. See Big compute architecture style.

Use leader election for coordination. In cases where you need to coordinate operations, make sure the coordinator does not become a single point of failure in the application. Using the Leader Election pattern, one instance is the leader at any time, and acts as the coordinator. If the leader fails, a new instance is elected to be the leader.

Design to scale out

11/19/2021 • 2 minutes to read • Edit Online

Design your application so that it can scale horizontally

A primary advantage of the cloud is elastic scaling — the ability to use as much capacity as you need, scaling out as load increases, and scaling in when the extra capacity is not needed. Design your application so that it can scale horizontally, adding or removing new instances as demand requires.

Recommendations

Avoid instance stickiness. Stickiness, or *session affinity*, is when requests from the same client are always routed to the same server. Stickiness limits the application's ability to scale out. For example, traffic from a high-volume user will not be distributed across instances. Causes of stickiness include storing session state in memory, and using machine-specific keys for encryption. Make sure that any instance can handle any request.

Identify bottlenecks. Scaling out isn't a magic fix for every performance issue. For example, if your backend database is the bottleneck, it won't help to add more web servers. Identify and resolve the bottlenecks in the system first, before throwing more instances at the problem. Stateful parts of the system are the most likely cause of bottlenecks.

Decompose workloads by scalability requirements. Applications often consist of multiple workloads, with different requirements for scaling. For example, an application might have a public-facing site and a separate administration site. The public site may experience sudden surges in traffic, while the administration site has a smaller, more predictable load.

Offload resource-intensive tasks. Tasks that require a lot of CPU or I/O resources should be moved to background jobs when possible, to minimize the load on the front end that is handling user requests.

Use built-in autoscaling features. Many Azure compute services have built-in support for autoscaling. If the application has a predictable, regular workload, scale out on a schedule. For example, scale out during business hours. Otherwise, if the workload is not predictable, use performance metrics such as CPU or request queue length to trigger autoscaling. For autoscaling best practices, see Autoscaling.

Consider aggressive autoscaling for critical workloads. For critical workloads, you want to keep ahead of demand. It's better to add new instances quickly under heavy load to handle the additional traffic, and then gradually scale back.

Design for scale in. Remember that with elastic scale, the application will have periods of scale in, when instances get removed. The application must gracefully handle instances being removed. Here are some ways to handle scalein:

- Listen for shutdown events (when available) and shut down cleanly.
- Clients/consumers of a service should support transient fault handling and retry.
- For long-running tasks, consider breaking up the work, using checkpoints or the Pipes and Filters pattern.
- Put work items on a queue so that another instance can pick up the work, if an instance is removed in the middle of processing.

Partition around limits

11/19/2021 • 2 minutes to read • Edit Online

Use partitioning to work around database, network, and compute limits

In the cloud, all services have limits in their ability to scale up. Azure service limits are documented in Azure subscription and service limits, quotas, and constraints. Limits include number of cores, database size, query throughput, and network throughput. If your system grows sufficiently large, you may hit one or more of these limits. Use partitioning to work around these limits.

There are many ways to partition a system, such as:

- Partition a database to avoid limits on database size, data I/O, or number of concurrent sessions.
- Partition a queue or message bus to avoid limits on the number of requests or the number of concurrent connections.
- Partition an App Service web app to avoid limits on the number of instances per App Service plan.

A database can be partitioned *horizontally*, *vertically*, or *functionally*.

- In horizontal partitioning, also called sharding, each partition holds data for a subset of the total data set. The partitions share the same data schema. For example, customers whose names start with A–M go into one partition, N–Z into another partition.
- In vertical partitioning, each partition holds a subset of the fields for the items in the data store. For example, put frequently accessed fields in one partition, and less frequently accessed fields in another.
- In functional partitioning, data is partitioned according to how it is used by each bounded context in the system. For example, store invoice data in one partition and product inventory data in another. The schemas are independent.

For more detailed guidance, see Data partitioning.

Recommendations

Partition different parts of the application. Databases are one obvious candidate for partitioning, but also consider storage, cache, queues, and compute instances.

Design the partition key to avoid hotspots. If you partition a database, but one shard still gets the majority of the requests, then you haven't solved your problem. Ideally, load gets distributed evenly across all the partitions. For example, hash by customer ID and not the first letter of the customer name, because some letters are more frequent. The same principle applies when partitioning a message queue. Pick a partition key that leads to an even distribution of messages across the set of queues. For more information, see Sharding.

Partition around Azure subscription and service limits. Individual components and services have limits, but there are also limits for subscriptions and resource groups. For very large applications, you might need to partition around those limits.

Partition at different levels. Consider a database server deployed on a VM. The VM has a VHD that is backed by Azure Storage. The storage account belongs to an Azure subscription. Notice that each step in the hierarchy has limits. The database server may have a connection pool limit. VMs have CPU and network limits. Storage has IOPS limits. The subscription has limits on the number of VM cores. Generally, it's easier to partition lower in the

hierarchy. Only large applications should need to partition at the subscription level.

Design for operations

11/19/2021 • 2 minutes to read • Edit Online

Design an application so that the operations team has the tools they need

The cloud has dramatically changed the role of the operations team. They are no longer responsible for managing the hardware and infrastructure that hosts the application. That said, operations is still a critical part of running a successful cloud application. Some of the important functions of the operations team include:

- Deployment
- Monitoring
- Escalation
- Incident response
- Security auditing

Robust logging and tracing are particularly important in cloud applications. Involve the operations team in design and planning, to ensure the application gives them the data and insight they need to be successful.

Recommendations

Make all things observable. Once a solution is deployed and running, logs and traces are your primary insight into the system. *Tracing* records a path through the system, and is useful to pinpoint bottlenecks, performance issues, and failure points. *Logging* captures individual events such as application state changes, errors, and exceptions. Log in production, or else you lose insight at the very times when you need it the most.

Instrument for monitoring. Monitoring gives insight into how well (or poorly) an application is performing, in terms of availability, performance, and system health. For example, monitoring tells you whether you are meeting your SLA. Monitoring happens during the normal operation of the system. It should be as close to real-time as possible, so that the operations staff can react to issues quickly. Ideally, monitoring can help avert problems before they lead to a critical failure. For more information, see Monitoring and diagnostics.

Instrument for root cause analysis. Root cause analysis is the process of finding the underlying cause of failures. It occurs after a failure has already happened.

Use distributed tracing. Use a distributed tracing system that is designed for concurrency, asynchrony, and cloud scale. Traces should include a correlation ID that flows across service boundaries. A single operation may involve calls to multiple application services. If an operation fails, the correlation ID helps to pinpoint the cause of the failure.

Standardize logs and metrics. The operations team will need to aggregate logs from across the various services in your solution. If every service uses its own logging format, it becomes difficult or impossible to get useful information from them. Define a common schema that includes fields such as correlation ID, event name, IP address of the sender, and so forth. Individual services can derive custom schemas that inherit the base schema, and contain additional fields.

Automate management tasks, including provisioning, deployment, and monitoring. Automating a task makes it repeatable and less prone to human errors.

Treat configuration as code. Check configuration files into a version control system, so that you can track and version your changes, and roll back if needed.

Use platform as a service (PaaS) options

11/19/2021 • 2 minutes to read • Edit Online

When possible, use platform as a service (PaaS) rather than infrastructure as a service (IaaS)

IaaS is like having a box of parts. You can build anything, but you have to assemble it yourself. PaaS options are easier to configure and administer. You don't need to provision VMs, set up VNets, manage patches and updates, and all of the other overhead associated with running software on a VM.

For example, suppose your application needs a message queue. You could set up your own messaging service on a VM, using something like RabbitMQ. But Azure Service Bus already provides reliable messaging as service, and it's simpler to set up. Just create a Service Bus namespace (which can be done as part of a deployment script) and then call Service Bus using the client SDK.

Of course, your application may have specific requirements that make an IaaS approach more suitable. However, even if your application is based on IaaS, look for places where it may be natural to incorporate PaaS options. These include cache, queues, and data storage.

INSTEAD OF RUNNING...	CONSIDER USING...
Active Directory	Azure Active Directory
Elasticsearch	Azure Search
Hadoop	HDInsight
IIS	App Service
MongoDB	Cosmos DB
Redis	Azure Cache for Redis
SQL Server	Azure SQL Database
File share	Azure NetApp Files

Please note that this is not meant to be an exhaustive list, but a subset of equivalent options.

Use the best data store for the job

11/19/2021 • 2 minutes to read • Edit Online

Pick the storage technology that is the best fit for your data and how it will be used

Gone are the days when you would just stick all of your data into a big relational SQL database. Relational databases are very good at what they do — providing ACID guarantees for transactions over relational data. But they come with some costs:

- Queries may require expensive joins.
- Data must be normalized and conform to a predefined schema (schema on write).
- Lock contention may impact performance.

In any large solution, it's likely that a single data store technology won't fill all your needs. Alternatives to relational databases include key/value stores, document databases, search engine databases, time series databases, column family databases, and graph databases. Each has pros and cons, and different types of data fit more naturally into one or another.

For example, you might store a product catalog in a document database, such as Cosmos DB, which allows for a flexible schema. In that case, each product description is a self-contained document. For queries over the entire catalog, you might index the catalog and store the index in Azure Search. Product inventory might go into a SQL database, because that data requires ACID guarantees.

Remember that data includes more than just the persisted application data. It also includes application logs, events, messages, and caches.

Recommendations

Don't use a relational database for everything. Consider other data stores when appropriate. See Choose the right data store.

Embrace polyglot persistence. In any large solution, it's likely that a single data store technology won't fill all your needs.

Consider the type of data. For example, put transactional data into SQL, put JSON documents into a document database, put telemetry data into a time series data base, put application logs in Elasticsearch, and put blobs in Azure Blob Storage.

Prefer availability over (strong) consistency. The CAP theorem implies that a distributed system must make trade-offs between availability and consistency. (Network partitions, the other leg of the CAP theorem, can never be completely avoided.) Often, you can achieve higher availability by adopting an *eventual consistency* model.

Consider the skillset of the development team. There are advantages to using polyglot persistence, but it's possible to go overboard. Adopting a new data storage technology requires a new set of skills. The development team must understand how to get the most out of the technology. They must understand appropriate usage patterns, how to optimize queries, tune for performance, and so on. Factor this in when considering storage technologies.

Use compensating transactions. A side effect of polyglot persistence is that single transaction might write data to multiple stores. If something fails, use compensating transactions to undo any steps that already

completed.

Look at bounded contexts. *Bounded context* is a term from domain driven design. A bounded context is an explicit boundary around a domain model, and defines which parts of the domain the model applies to. Ideally, a bounded context maps to a subdomain of the business domain. The bounded contexts in your system are a natural place to consider polyglot persistence. For example, "products" may appear in both the Product Catalog subdomain and the Product Inventory subdomain, but it's very likely that these two subdomains have different requirements for storing, updating, and querying products.

Design for evolution

11/19/2021 • 3 minutes to read • Edit Online

An evolutionary design is key for continuous innovation

All successful applications change over time, whether to fix bugs, add new features, bring in new technologies, or make existing systems more scalable and resilient. If all the parts of an application are tightly coupled, it becomes very hard to introduce changes into the system. A change in one part of the application may break another part, or cause changes to ripple through the entire codebase.

This problem is not limited to monolithic applications. An application can be decomposed into services, but still exhibit the sort of tight coupling that leaves the system rigid and brittle. But when services are designed to evolve, teams can innovate and continuously deliver new features.

Microservices are becoming a popular way to achieve an evolutionary design, because they address many of the considerations listed here.

Recommendations

Enforce high cohesion and loose coupling. A service is *cohesive* if it provides functionality that logically belongs together. Services are *loosely coupled* if you can change one service without changing the other. High cohesion generally means that changes in one function will require changes in other related functions. If you find that updating a service requires coordinated updates to other services, it may be a sign that your services are not cohesive. One of the goals of domain-driven design (DDD) is to identify those boundaries.

Encapsulate domain knowledge. When a client consumes a service, the responsibility for enforcing the business rules of the domain should not fall on the client. Instead, the service should encapsulate all of the domain knowledge that falls under its responsibility. Otherwise, every client has to enforce the business rules, and you end up with domain knowledge spread across different parts of the application.

Use asynchronous messaging. Asynchronous messaging is a way to decouple the message producer from the consumer. The producer does not depend on the consumer responding to the message or taking any particular action. With a pub/sub architecture, the producer may not even know who is consuming the message. New services can easily consume the messages without any modifications to the producer.

Don't build domain knowledge into a gateway. Gateways can be useful in a microservices architecture, for things like request routing, protocol translation, load balancing, or authentication. However, the gateway should be restricted to this sort of infrastructure functionality. It should not implement any domain knowledge, to avoid becoming a heavy dependency.

Expose open interfaces. Avoid creating custom translation layers that sit between services. Instead, a service should expose an API with a well-defined API contract. The API should be versioned, so that you can evolve the API while maintaining backward compatibility. That way, you can update a service without coordinating updates to all of the upstream services that depend on it. Public facing services should expose a RESTful API over HTTP. Backend services might use an RPC-style messaging protocol for performance reasons.

Design and test against service contracts. When services expose well-defined APIs, you can develop and test against those APIs. That way, you can develop and test an individual service without spinning up all of its dependent services. (Of course, you would still perform integration and load testing against the real services.)

Abstract infrastructure away from domain logic. Don't let domain logic get mixed up with infrastructure-related functionality, such as messaging or persistence. Otherwise, changes in the domain logic will require

updates to the infrastructure layers and vice versa.

Offload cross-cutting concerns to a separate service. For example, if several services need to authenticate requests, you could move this functionality into its own service. Then you could evolve the authentication service — for example, by adding a new authentication flow — without touching any of the services that use it.

Deploy services independently. When the DevOps team can deploy a single service independently of other services in the application, updates can happen more quickly and safely. Bug fixes and new features can be rolled out at a more regular cadence. Design both the application and the release process to support independent updates.

Build for the needs of the business

11/19/2021 • 2 minutes to read • Edit Online

Every design decision must be justified by a business requirement

This design principle may seem obvious, but it's crucial to keep in mind when designing a solution. Do you anticipate millions of users, or a few thousand? Is a one-hour application outage acceptable? Do you expect large bursts in traffic or a predictable workload? Ultimately, every design decision must be justified by a business requirement.

Recommendations

Define business objectives, including the recovery time objective (RTO), recovery point objective (RPO), and maximum tolerable outage (MTO). These numbers should inform decisions about the architecture. For example, to achieve a low RTO, you might implement automated failover to a secondary region. But if your solution can tolerate a higher RTO, that degree of redundancy might be unnecessary.

Document service level agreements (SLA) and service level objectives (SLO), including availability and performance metrics. You might build a solution that delivers 99.95% availability. Is that enough? The answer is a business decision.

Model the application around the business domain. Start by analyzing the business requirements. Use these requirements to model the application. Consider using a domain-driven design (DDD) approach to create domain models that reflect the business processes and use cases.

Capture both functional and nonfunctional requirements. Functional requirements let you judge whether the application does the right thing. Nonfunctional requirements let you judge whether the application does those things *well*. In particular, make sure that you understand your requirements for scalability, availability, and latency. These requirements will influence design decisions and choice of technology.

Decompose by workload. The term "workload" in this context means a discrete capability or computing task, which can be logically separated from other tasks. Different workloads may have different requirements for availability, scalability, data consistency, and disaster recovery.

Plan for growth. A solution might meet your current needs, in terms of number of users, volume of transactions, data storage, and so forth. However, a robust application can handle growth without major architectural changes. See Design to scale out and Partition around limits. Also consider that your business model and business requirements will likely change over time. If an application's service model and data models are too rigid, it becomes hard to evolve the application for new use cases and scenarios. See Design for evolution.

Manage costs. In a traditional on-premises application, you pay upfront for hardware as a capital expenditure. In a cloud application, you pay for the resources that you consume. Make sure that you understand the pricing model for the services that you consume. The total cost will include network bandwidth usage, storage, IP addresses, service consumption, and other factors. For more information, see Azure pricing. Also consider your operations costs. In the cloud, you don't have to manage the hardware or other infrastructure, but you still need to manage your applications, including DevOps, incident response, disaster recovery, and so forth.

Choose an Azure compute service for your application

11/19/2021 • 7 minutes to read • Edit Online

Azure offers a number of ways to host your application code. The term *compute* refers to the hosting model for the computing resources that your application runs on. The following flowchart will help you to choose a compute service for your application.

If your application consists of multiple workloads, evaluate each workload separately. A complete solution may incorporate two or more compute services.

Choose a candidate service

Use the following flowchart to select a candidate compute service.

Definitions:

- **"Lift and shift"** is a strategy for migrating a workload to the cloud without redesigning the application or making code changes. Also called *rehosting*. For more information, see Azure migration and modernization

center.

- **Cloud optimized** is a strategy for migrating to the cloud by refactoring an application to take advantage of cloud-native features and capabilities.

The output from this flowchart is a **starting point** for consideration. Next, perform a more detailed evaluation of the service to see if it meets your needs.

This article includes several tables which may help you to make these tradeoff decisions. Based on this analysis, you may find that the initial candidate isn't suitable for your particular application or workload. In that case, expand your analysis to include other compute services.

> **NOTE**
> Learn more about reviewing your compute requirements for cloud adoption, in the Microsoft Cloud Adoption Framework for Azure.

Understand the basic features

If you're not familiar with the Azure service selected in the previous step, read the overview documentation to understand the basics of the service.

- App Service. A managed service for hosting web apps, mobile app back ends, RESTful APIs, or automated business processes.
- Azure Spring Cloud. A managed service designed and optimized for hosting Spring Boot apps.
- Azure Kubernetes Service (AKS). A managed Kubernetes service for running containerized applications.
- Batch. A managed service for running large-scale parallel and high-performance computing (HPC) applications
- Container Instances. The fastest and simplest way to run a container in Azure, without having to provision any virtual machines and without having to adopt a higher-level service.
- Functions. A managed FaaS service.
- Service Fabric. A distributed systems platform that can run in many environments, including Azure or on premises.
- Virtual machines. Deploy and manage VMs inside an Azure virtual network.

Understand the hosting models

Cloud services, including Azure services, generally fall into three categories: IaaS, PaaS, or FaaS. (There is also SaaS, software-as-a-service, which is out of scope for this article.) It's useful to understand the differences.

Infrastructure-as-a-Service (IaaS) lets you provision individual VMs along with the associated networking and storage components. Then you deploy whatever software and applications you want onto those VMs. This model is the closest to a traditional on-premises environment, except that Microsoft manages the infrastructure. You still manage the individual VMs.

Platform-as-a-Service (PaaS) provides a managed hosting environment, where you can deploy your application without needing to manage VMs or networking resources. Azure App Service is a PaaS service.

Functions-as-a-Service (FaaS) goes even further in removing the need to worry about the hosting environment. In a FaaS model, you simply deploy your code and the service automatically runs it. Azure Functions is a FaaS service.

> **NOTE**
> Azure Functions is an Azure serverless compute offering. You may read Choose the right integration and automation services in Azure to know how this service compares with other Azure serverless offerings, such as Logic Apps which provides serverless workflows.

There is a spectrum from IaaS to pure PaaS. For example, Azure VMs can autoscale by using virtual machine scale sets. This automatic scaling capability isn't strictly PaaS, but it's the type of management feature found in PaaS services.

In general, there is a tradeoff between control and ease of management. IaaS gives the most control, flexibility, and portability, but you have to provision, configure and manage the VMs and network components you create. FaaS services automatically manage nearly all aspects of running an application. PaaS services fall somewhere in between.

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Application composition	Agnostic	Applications, containers	Applications, microservices	Services, guest executables, containers	Functions	Containers	Containers	Scheduled jobs
Density	Agnostic	Multiple apps per instance via app service plans	Multiple apps per service instance	Multiple services per VM	Serverless [1]	Multiple containers per node	No dedicated instances	Multiple apps per VM
Minimum number of nodes	1 [2]	1	2	5 [3]	Serverless [1]	3 [3]	No dedicated nodes	1 [4]
State management	Stateless or Stateful	Stateless	Stateless	Stateless or stateful	Stateless	Stateless or Stateful	Stateless	Stateless
Web hosting	Agnostic	Built in	Built in	Agnostic	Not applicable	Agnostic	Agnostic	No
Can be deployed to dedicated VNet?	Supported	Supported[5]	Supported	Supported	Supported [5]	Supported	Supported	Supported
Hybrid connectivity	Supported	Supported [6]	Supported	Supported	Supported [7]	Supported	Not supported	Supported

Notes

1. If using Consumption plan. If using App Service plan, functions run on the VMs allocated for your App

Service plan. See Choose the correct service plan for Azure Functions.

2. Higher SLA with two or more instances.
3. Recommended for production environments.
4. Can scale down to zero after job completes.
5. Requires App Service Environment (ASE).
6. Use Azure App Service Hybrid Connections.
7. Requires App Service plan or Azure Functions Premium plan.

DevOps

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Local debugging	Agnostic	IIS Express, others [1]	Visual Studio Code, Intellij, Eclipse	Local node cluster	Visual Studio or Azure Functions CLI	Minikube, others	Local container runtime	Not supported
Programming model	Agnostic	Web and API applications, WebJobs for background tasks	Spring Boot, Steeltoe	Guest executable, Service model, Actor model, Containers	Functions with triggers	Agnostic	Agnostic	Command line application
Application update	No built-in support	Deployment slots	Rolling upgrade, Blue-green deployment	Rolling upgrade (per service)	Deployment slots	Rolling update	Not applicable	

Notes

1. Options include IIS Express for ASP.NET or node.js (iisnode); PHP web server; Azure Toolkit for IntelliJ, Azure Toolkit for Eclipse. App Service also supports remote debugging of deployed web app.
2. See Resource Manager providers, regions, API versions and schemas.

Scalability

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Autoscaling	Virtual machine scale sets	Built-in service	Built-in service	Virtual machine scale sets	Built-in service	Pod auto-scaling[1], cluster auto-scaling[2]	Not supported	N/A

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Load balancer	Azure Load Balancer	Integrated	Integrated	Azure Load Balancer	Integrated	Azure Load Balancer or Application Gateway	No built-in support	Azure Load Balancer
Scale limit[3]	Platform image: 1000 nodes per scale set, Custom image: 600 nodes per scale set	30 instances, 100 with App Service Environment	500 app instances in Standard	100 nodes per scale set	200 instances per Function app	100 nodes per cluster (default limit)	20 container groups per subscription (default limit).	20 core limit (default limit).

Notes

1. See Autoscale pods.
2. See Automatically scale a cluster to meet application demands on Azure Kubernetes Service (AKS).
3. See Azure subscription and service limits, quotas, and constraints.

Availability

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
SLA	SLA for Virtual Machines	SLA for App Service	SLA for Azure Spring Cloud	SLA for Service Fabric	SLA for Functions	SLA for AKS	SLA for Container Instances	SLA for Azure Batch
Multi region failover	Traffic manager	Traffic manager	Traffic manager, Multi-Region Cluster	Azure Front Door	Traffic manager	Not supported	Not Supported	

For guided learning on Service Guarantees, review Core Cloud Services - Azure architecture and service guarantees.

Security

Review and understand the available security controls and visibility for each service

- App Service
- App Spring Cloud
- Azure Kubernetes Service
- Batch

- Container Instances
- Functions
- Service Fabric
- Virtual machine - Windows
- Virtual machine - LINUX

Other criteria

CRITERIA	VIRTUAL MACHINES	APP SERVICE	APP SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
SSL	Configured in VM	Supported	Supported	Supported	Supported	Ingress controller	Use sidecar container	Supported
Cost	Windows, Linux	App Service pricing	Azure Spring Cloud pricing	Service Fabric pricing	Azure Functions pricing	AKS pricing	Container Instances pricing	Azure Batch pricing
Suitable architecture styles	N-Tier, Big compute (HPC)	Web-Queue-Worker, N-Tier	Spring Boot, Microservices	Microservices, Event-driven architecture	Microservices, Event-driven architecture	Microservices, Event-driven architecture	Microservices, task automation, batch jobs	Big compute (HPC)

The output from this flowchart is a **starting point** for consideration. Next, perform a more detailed evaluation of the service to see if it meets your needs.

Consider limits and cost

Perform a more detailed evaluation looking at the following aspects of the service:

- Service limits
- Cost
- SLA
- Regional availability
- Compute comparison tables

Next steps

- Core Cloud Services - Azure compute options. This Microsoft Learn module explores how compute services can solve common business needs.

Choose a Kubernetes at the edge compute option

11/19/2021 • 6 minutes to read • Edit Online

This document discusses the trade-offs for various options available for extending compute on the edge. The following considerations for each Kubernetes option are covered:

- **Operational cost.** The expected labor required to maintain and operate the Kubernetes clusters.
- **Ease of configuration.** The level of difficulty to configure and deploy a Kubernetes cluster.
- **Flexibility.** A measure of how adaptable the Kubernetes option is to integrate a customized configuration with existing infrastructure at the edge.
- **Mixed node.** Ability to run a Kubernetes cluster with both Linux and Windows nodes.

Assumptions

- You are a cluster operator looking to understand different options for running Kubernetes at the edge and managing clusters in Azure.
- You have a good understanding of existing infrastructure and any other infrastructure requirements, including storage and networking requirements.

After reading this document, you'll be in a better position to identify which option best fits your scenario and the environment required.

Kubernetes choices at a glance

	OPERATIONAL COST	EASE OF CONFIGURATION	FLEXIBILITY	MIXED NODE	SUMMARY
Bare-metal Kubernetes	High**	Difficult**	High**	Yes	A ground-up configuration on any available infrastructure at location with the option to use Azure Arc for added Azure capabilities.
K8s on Azure Stack Edge Pro	Low	Easy	Low	Linux only	Kubernetes deployed on Azure Stack Edge appliance deployed at location.
AKS on HCI	Low	Easy	Medium	Yes	AKS deployed on Azure Stack HCI or Windows Server 2019.

*Other managed edge platforms (OpenShift, Tanzu, and so on) aren't in scope for this document.

**These values are based on using *kubeadm*, for the sake of simplicity. Different options for running bare-metal Kubernetes at the edge would alter the rating in these categories.

Bare-metal Kubernetes

Ground-up configuration of Kubernetes using tools like kubeadm on any underlying infrastructure.

The biggest constraints for bare-metal Kubernetes are around the specific needs and requirements of the organization. The opportunity to use any distribution, networking interface, and plugin means higher complexity and operational cost. But this offers the most flexible option for customizing your cluster.

Scenario

Often, *edge* locations have specific requirements for running Kubernetes clusters that aren't met with the other Azure solutions described in this document. Meaning this option is typically best for those unable to use managed services due to unsupported existing infrastructure, or those who seek to have maximum control of their clusters.

- This option can be especially difficult for those who are new to Kubernetes. This isn't uncommon for organizations looking to run edge clusters. Options like MicroK8s or k3s aim to flatten that learning curve.
- It's important to understand any underlying infrastructure and any integration that is expected to take place up front. This will help to narrow down viable options and to identify any gaps with the open-source tooling and/or plugins.
- Enabling clusters with Azure Arc presents a simple way to manage your cluster from Azure alongside other resources. This also brings other Azure capabilities to your cluster, including Azure Policy, Azure Monitor, Microsoft Defender for Cloud, and other services.
- Because cluster configuration isn't trivial, it's especially important to be mindful of CI/CD. Tracking and acting on upstream changes of various plugins, and making sure those changes don't affect the health of your cluster, becomes a direct responsibility. It's important for you to have a strong CI/CD solution, strong testing, and monitoring in place.

Tooling options

Cluster bootstrap:

- kubeadm: Kubernetes tool for creating ground-up Kubernetes clusters. Good for standard compute resources (Linux/Windows).
- MicroK8s: Simplified administration and configuration ("*LowOps*"), conformant Kubernetes by Canonical.
- k3s: Certified Kubernetes distribution built for Internet of Things (IoT) and edge computing.

Storage:

- Explore available CSI drivers: Many options are available to fit your requirements from cloud to local file shares.

Networking:

- A full list of available add-ons can be found here: Networking add-ons. Some popular options include Flannel, a simple overlay network, and Calico, which provides a full networking stack.

Considerations

Operational cost:

- Without the support that comes with managed services, it's up to the organization to maintain and operate

the cluster as a whole (storage, networking, upgrades, observability, application management). The operational cost is considered high.

Ease of configuration:

- Evaluating the many open-source options at every stage of configuration whether its networking, storage, or monitoring options is inevitable and can become complex. Requires more consideration for configuring a CI/CD for cluster configuration. Because of these concerns, the ease of configuration is considered difficult.

Flexibility:

- With the ability to use any open-source tool or plugin without any provider restrictions, bare-metal Kubernetes is highly flexible.

Kubernetes on Azure Stack Edge

Kubernetes cluster (a master VM and a worker VM) configured and deployed for you on your Azure Stack Edge Pro device.

Azure Stack Edge Pro devices deliver Azure capabilities like compute, storage, networking, and hardware-accelerated machine learning (ML) to any edge location. Kubernetes clusters can be created once the compute role is enabled on any of the Pro-GPU, Pro-R, and Mini-R devices. Managing upgrades of the Kubernetes cluster can be done using standard updates available for the device.

Scenario

Ideal for those with existing (Linux) IoT workloads or upgrading their compute for ML at the edge. This is a good option when it isn't necessary to have more granular control over the clusters.

- Admin permissions aren't granted by default. Although you can work with the product group to make certain exceptions, this makes it difficult to have finer control of your cluster.
- There is an extra cost if there isn't already an Azure Stack Edge device. Explore Azure Stack Edge devices and see if any fit your compute requirements.
- Calico, MetalLB, and CoreDNS are installed for Kubernetes networking on the device.
- Only `Linux` workloads are supported at this time.
- In addition to Kubernetes, Azure Stack Edge also comes with the IoT runtime, which means that workloads may also be deployed to your Azure Stack Edge clusters via IoT Edge.
- Support for two node clusters isn't currently available. This effectively means that this option is *not* a highly available (HA) solution.

Considerations

Operational cost:

- With the support that comes with the device, operational cost is minimal and is scoped to workload management.

Ease of configuration:

- Pre-configured and well-documented Kubernetes cluster deployment simplifies the configuration required compared to bare-metal Kubernetes.

Flexibility:

- Configuration is already set, and Admin permissions aren't granted by default. Product group involvement may be required beyond basic configuration, and the underlying infrastructure must be an Azure Stack Edge

Pro device, making this a less flexible option.

AKS on HCI

Note: This option is currently in **preview**.

AKS-HCI is a set of predefined settings and configurations that is used to deploy one or more Kubernetes clusters (with Windows Admin Center or PowerShell modules) on a multi-node cluster running either Windows Server 2019 Datacenter or Azure Stack HCI 20H2.

Scenario

Ideal for those who want a simplified and streamlined way to get a Microsoft-supported cluster on compatible devices (Azure Stack HCI or Windows Server 2019 Datacenter). Operations and configuration complexity are reduced at the expense of the flexibility when compared to the bare-metal Kubernetes option.

Considerations

At the time of this writing, the preview comes with many limitations (permissions, networking limitations, large compute requirements, and documentation gaps). Purposes other than evaluation and development are discouraged that this time.

Operational cost:

- Microsoft-supported cluster minimizes operational costs.

Ease of configuration:

- Pre-configured and well-documented Kubernetes cluster deployment simplifies the configuration required compared to bare-metal Kubernetes.

Flexibility:

- Cluster configuration itself is set, but Admin permissions are granted. The underlying infrastructure must either be Azure Stack HCI or Windows Server 2019. This option is more flexible than Kubernetes on Azure Stack Edge and less flexible than bare-metal Kubernetes.

Next steps

For more information, see the following articles:

- What is Azure IoT Edge
- Kubernetes on your Azure Stack Edge Pro GPU device
- Use IoT Edge module to run a Kubernetes stateless application on your Azure Stack Edge Pro GPU device
- Deploy a Kubernetes stateless application via kubectl on your Azure Stack Edge Pro GPU device
- AI at the edge with Azure Stack Hub
- Building a CI/CD pipeline for microservices on Kubernetes
- Use Kubernetes dashboard to monitor your Azure Stack Edge Pro GPU device

Understand data store models

11/19/2021 • 12 minutes to read • Edit Online

Modern business systems manage increasingly large volumes of heterogeneous data. This heterogeneity means that a single data store is usually not the best approach. Instead, it's often better to store different types of data in different data stores, each focused toward a specific workload or usage pattern. The term *polyglot persistence* is used to describe solutions that use a mix of data store technologies. Therefore, it's important to understand the main storage models and their tradeoffs.

Selecting the right data store for your requirements is a key design decision. There are literally hundreds of implementations to choose from among SQL and NoSQL databases. Data stores are often categorized by how they structure data and the types of operations they support. This article describes several of the most common storage models. Note that a particular data store technology may support multiple storage models. For example, a relational database management systems (RDBMS) may also support key/value or graph storage. In fact, there is a general trend for so-called *multi-model* support, where a single database system supports several models. But it's still useful to understand the different models at a high level.

Not all data stores in a given category provide the same feature-set. Most data stores provide server-side functionality to query and process data. Sometimes this functionality is built into the data storage engine. In other cases, the data storage and processing capabilities are separated, and there may be several options for processing and analysis. Data stores also support different programmatic and management interfaces.

Generally, you should start by considering which storage model is best suited for your requirements. Then consider a particular data store within that category, based on factors such as feature set, cost, and ease of management.

> **NOTE**
>
> Learn more about identifying and reviewing your data service requirements for cloud adoption, in the Microsoft Cloud Adoption Framework for Azure. Likewise, you can also learn about selecting storage tools and services.

Relational database management systems

Relational databases organize data as a series of two-dimensional tables with rows and columns. Most vendors provide a dialect of the Structured Query Language (SQL) for retrieving and managing data. An RDBMS typically implements a transactionally consistent mechanism that conforms to the ACID (Atomic, Consistent, Isolated, Durable) model for updating information.

An RDBMS typically supports a schema-on-write model, where the data structure is defined ahead of time, and all read or write operations must use the schema.

This model is very useful when strong consistency guarantees are important — where all changes are atomic, and transactions always leave the data in a consistent state. However, an RDBMS generally can't scale out horizontally without sharding the data in some way. Also, the data in an RDBMS must be normalized, which isn't appropriate for every data set.

Azure services

- Azure SQL Database | (Security Baseline)
- Azure Database for MySQL | (Security Baseline)
- Azure Database for PostgreSQL | (Security Baseline)

- Azure Database for MariaDB | (Security Baseline)

Workload

- Records are frequently created and updated.
- Multiple operations have to be completed in a single transaction.
- Relationships are enforced using database constraints.
- Indexes are used to optimize query performance.

Data type

- Data is highly normalized.
- Database schemas are required and enforced.
- Many-to-many relationships between data entities in the database.
- Constraints are defined in the schema and imposed on any data in the database.
- Data requires high integrity. Indexes and relationships need to be maintained accurately.
- Data requires strong consistency. Transactions operate in a way that ensures all data are 100% consistent for all users and processes.
- Size of individual data entries is small to medium-sized.

Examples

- Inventory management
- Order management
- Reporting database
- Accounting

Key/value stores

A key/value store associates each data value with a unique key. Most key/value stores only support simple query, insert, and delete operations. To modify a value (either partially or completely), an application must overwrite the existing data for the entire value. In most implementations, reading or writing a single value is an atomic operation.

An application can store arbitrary data as a set of values. Any schema information must be provided by the application. The key/value store simply retrieves or stores the value by key.

Key/value stores are highly optimized for applications performing simple lookups, but are less suitable if you need to query data across different key/value stores. Key/value stores are also not optimized for querying by value.

A single key/value store can be extremely scalable, as the data store can easily distribute data across multiple nodes on separate machines.

Azure services

- Azure Cosmos DB Table API, etcd API (preview), and SQL API | (Cosmos DB Security Baseline)
- Azure Cache for Redis | (Security Baseline)
- Azure Table Storage | (Security Baseline)

Workload

- Data is accessed using a single key, like a dictionary.
- No joins, lock, or unions are required.
- No aggregation mechanisms are used.
- Secondary indexes are generally not used.

Data type

- Each key is associated with a single value.
- There is no schema enforcement.
- No relationships between entities.

Examples

- Data caching
- Session management
- User preference and profile management
- Product recommendation and ad serving

Document databases

A document database stores a collection of *documents*, where each document consists of named fields and data. The data can be simple values or complex elements such as lists and child collections. Documents are retrieved by unique keys.

Typically, a document contains the data for single entity, such as a customer or an order. A document may contain information that would be spread across several relational tables in an RDBMS. Documents don't need to have the same structure. Applications can store different data in documents as business requirements change.

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

Azure service

- Azure Cosmos DB SQL API | (Cosmos DB Security Baseline)

Workload

- Insert and update operations are common.
- No object-relational impedance mismatch. Documents can better match the object structures used in application code.
- Individual documents are retrieved and written as a single block.

- Data requires index on multiple fields.

Data type

- Data can be managed in de-normalized way.
- Size of individual document data is relatively small.
- Each document type can use its own schema.
- Documents can include optional fields.
- Document data is semi-structured, meaning that data types of each field are not strictly defined.

Examples

- Product catalog
- Content management
- Inventory management

Graph databases

A graph database stores two types of information, nodes and edges. Edges specify relationships between nodes. Nodes and edges can have properties that provide information about that node or edge, similar to columns in a table. Edges can also have a direction indicating the nature of the relationship.

Graph databases can efficiently perform queries across the network of nodes and edges and analyze the relationships between entities. The following diagram shows an organization's personnel database structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the departments in which employees work.

This structure makes it straightforward to perform queries such as "Find all employees who report directly or indirectly to Sarah" or "Who works in the same department as John?" For large graphs with lots of entities and relationships, you can perform very complex analyses very quickly. Many graph databases provide a query language that you can use to traverse a network of relationships efficiently.

Azure services

- Azure Cosmos DB Gremlin API | (Security Baseline)
- SQL Server | (Security Baseline)

Workload

- Complex relationships between data items involving many hops between related data items.
- The relationship between data items are dynamic and change over time.
- Relationships between objects are first-class citizens, without requiring foreign-keys and joins to traverse.

Data type

- Nodes and relationships.
- Nodes are similar to table rows or JSON documents.
- Relationships are just as important as nodes, and are exposed directly in the query language.
- Composite objects, such as a person with multiple phone numbers, tend to be broken into separate, smaller nodes, combined with traversable relationships

Examples

- Organization charts
- Social graphs
- Fraud detection
- Recommendation engines

Data analytics

Data analytics stores provide massively parallel solutions for ingesting, storing, and analyzing data. The data is distributed across multiple servers to maximize scalability. Large data file formats such as delimiter files (CSV), parquet, and ORC are widely used in data analytics. Historical data is typically stored in data stores such as blob storage or Azure Data Lake Storage Gen2, which are then accessed by Azure Synapse, Databricks, or HDInsight as external tables. A typical scenario using data stored as parquet files for performance, is described in the article Use external tables with Synapse SQL.

Azure services

- Azure Synapse Analytics | (Security Baseline)
- Azure Data Lake | (Security Baseline)
- Azure Data Explorer | (Security Baseline)
- Azure Analysis Services
- HDInsight | (Security Baseline)
- Azure Databricks | (Security Baseline)

Workload

- Data analytics
- Enterprise BI

Data type

- Historical data from multiple sources.
- Usually denormalized in a "star" or "snowflake" schema, consisting of fact and dimension tables.
- Usually loaded with new data on a scheduled basis.
- Dimension tables often include multiple historic versions of an entity, referred to as a *slowly changing dimension*.

Examples

- Enterprise data warehouse

Column-family databases

A column-family database organizes data into rows and columns. In its simplest form, a column-family database can appear very similar to a relational database, at least conceptually. The real power of a column-family

database lies in its denormalized approach to structuring sparse data.

You can think of a column-family database as holding tabular data with rows and columns, but the columns are divided into groups known as *column families*. Each column family holds a set of columns that are logically related together and are typically retrieved or manipulated as a unit. Other data that is accessed separately can be stored in separate column families. Within a column family, new columns can be added dynamically, and rows can be sparse (that is, a row doesn't need to have a value for every column).

The following diagram shows an example with two column families, `Identity` and `Contact Info`. The data for a single entity has the same row key in each column-family. This structure, where the rows for any given object in a column family can vary dynamically, is an important benefit of the column-family approach, making this form of data store highly suited for storing structured, volatile data.

CustomerID	Column Family: Identity
001	**First name:** Mu Bae **Last name:** Min
002	**First name:** Francisco **Last name:** Vila Nova **Suffix:** Jr.
003	**First name:** Lena **Last name:** Adamcyz **Title:** Dr.

CustomerID	Column Family: Contact Info
001	**Phone number:** 555-0100 **Email:** someone@example.com
002	**Email:** vilanova@contoso.com
003	**Phone number:** 555-0120

Unlike a key/value store or a document database, most column-family databases store data in key order, rather than by computing a hash. Many implementations allow you to create indexes over specific columns in a column-family. Indexes let you retrieve data by columns value, rather than row key.

Read and write operations for a row are usually atomic with a single column-family, although some implementations provide atomicity across the entire row, spanning multiple column-families.

Azure services

- Azure Cosmos DB Cassandra API | (Security Baseline)
- HBase in HDInsight | (Security Baseline)

Workload

- Most column-family databases perform write operations extremely quickly.
- Update and delete operations are rare.
- Designed to provide high throughput and low-latency access.
- Supports easy query access to a particular set of fields within a much larger record.
- Massively scalable.

Data type

- Data is stored in tables consisting of a key column and one or more column families.
- Specific columns can vary by individual rows.
- Individual cells are accessed via get and put commands
- Multiple rows are returned using a scan command.

Examples

- Recommendations
- Personalization
- Sensor data
- Telemetry
- Messaging
- Social media analytics

- Web analytics
- Activity monitoring
- Weather and other time-series data

Search Engine Databases

A search engine database allows applications to search for information held in external data stores. A search engine database can index massive volumes of data and provide near real-time access to these indexes.

Indexes can be multi-dimensional and may support free-text searches across large volumes of text data. Indexing can be performed using a pull model, triggered by the search engine database, or using a push model, initiated by external application code.

Searching can be exact or fuzzy. A fuzzy search finds documents that match a set of terms and calculates how closely they match. Some search engines also support linguistic analysis that can return matches based on synonyms, genre expansions (for example, matching `dogs` to `pets`), and stemming (matching words with the same root).

Azure service

- Azure Search | (Security Baseline)

Workload

- Data indexes from multiple sources and services.
- Queries are ad-hoc and can be complex.
- Full text search is required.
- Ad hoc self-service query is required.

Data type

- Semi-structured or unstructured text
- Text with reference to structured data

Examples

- Product catalogs
- Site search
- Logging

Time series databases

Time series data is a set of values organized by time. Time series databases typically collect large amounts of data in real time from a large number of sources. Updates are rare, and deletes are often done as bulk operations. Although the records written to a time-series database are generally small, there are often a large number of records, and total data size can grow rapidly.

Azure service

- Azure Time Series Insights

Workload

- Records are generally appended sequentially in time order.
- An overwhelming proportion of operations (95-99%) are writes.
- Updates are rare.
- Deletes occur in bulk, and are made to contiguous blocks or records.
- Data is read sequentially in either ascending or descending time order, often in parallel.

Data type

- A timestamp is used as the primary key and sorting mechanism.
- Tags may define additional information about the type, origin, and other information about the entry.

Examples

- Monitoring and event telemetry.
- Sensor or other IoT data.

Object storage

Object storage is optimized for storing and retrieving large binary objects (images, files, video and audio streams, large application data objects and documents, virtual machine disk images). Large data files are also popularly used in this model, for example, delimiter file (CSV), parquet, and ORC. Object stores can manage extremely large amounts of unstructured data.

Azure service

- Azure Blob Storage | (Security Baseline)
- Azure Data Lake Storage Gen2 | (Security Baseline)

Workload

- Identified by key.
- Content is typically an asset such as a delimiter, image, or video file.
- Content must be durable and external to any application tier.

Data type

- Data size is large.
- Value is opaque.

Examples

- Images, videos, office documents, PDFs
- Static HTML, JSON, CSS
- Log and audit files
- Database backups

Shared files

Sometimes, using simple flat files can be the most effective means of storing and retrieving information. Using file shares enables files to be accessed across a network. Given appropriate security and concurrent access control mechanisms, sharing data in this way can enable distributed services to provide highly scalable data access for performing basic, low-level operations such as simple read and write requests.

Azure service

- Azure Files | (Security Baseline)

Workload

- Migration from existing apps that interact with the file system.
- Requires SMB interface.

Data type

- Files in a hierarchical set of folders.
- Accessible with standard I/O libraries.

Examples

- Legacy files

- Shared content accessible among a number of VMs or app instances

Aided with this understanding of different data storage models, the next step is to evaluate your workload and application, and decide which data store will meet your specific needs. Use the data storage decision tree to help with this process.

Select an Azure data store for your application

11/19/2021 • 2 minutes to read • Edit Online

Azure offers a number of managed data storage solutions, each providing different features and capabilities. This article will help you to choose a managed data store for your application.

If your application consists of multiple workloads, evaluate each workload separately. A complete solution may incorporate multiple data stores.

Select a candidate

Use the following flowchart to select a candidate Azure managed data store.

The output from this flowchart is a **starting point** for consideration. Next, perform a more detailed evaluation of the data store to see if it meets your needs. Refer to Criteria for choosing a data store to aid in this evaluation.

Choose specialized storage

Alternative database solutions often require specific storage solutions. For example, SAP HANA on VMs often employs Azure NetApp Files as its underlying storage solution. Evaluate your vendor's requirements to find an appropriate storage solution to meet your database's requirements. For more information about selecting a storage solution, see Review your storage options.

Criteria for choosing a data store

11/19/2021 • 3 minutes to read • Edit Online

This article describes the comparison criteria you should use when evaluating a data store. The goal is to help you determine which data storage types can meet your solution's requirements.

General considerations

Keep the following considerations in mind when making your selection.

Functional requirements

- **Data format**. What type of data are you intending to store? Common types include transactional data, JSON objects, telemetry, search indexes, or flat files.
- **Data size**. How large are the entities you need to store? Will these entities need to be maintained as a single document, or can they be split across multiple documents, tables, collections, and so forth?
- **Scale and structure**. What is the overall amount of storage capacity you need? Do you anticipate partitioning your data?
- **Data relationships**. Will your data need to support one-to-many or many-to-many relationships? Are relationships themselves an important part of the data? Will you need to join or otherwise combine data from within the same dataset, or from external datasets?
- **Consistency model**. How important is it for updates made in one node to appear in other nodes, before further changes can be made? Can you accept eventual consistency? Do you need ACID guarantees for transactions?
- **Schema flexibility**. What kind of schemas will you apply to your data? Will you use a fixed schema, a schema-on-write approach, or a schema-on-read approach?
- **Concurrency**. What kind of concurrency mechanism do you want to use when updating and synchronizing data? Will the application perform many updates that could potentially conflict. If so, you may require record locking and pessimistic concurrency control. Alternatively, can you support optimistic concurrency controls? If so, is simple timestamp-based concurrency control enough, or do you need the added functionality of multi-version concurrency control?
- **Data movement**. Will your solution need to perform ETL tasks to move data to other stores or data warehouses?
- **Data lifecycle**. Is the data write-once, read-many? Can it be moved into cool or cold storage?
- **Other supported features**. Do you need any other specific features, such as schema validation, aggregation, indexing, full-text search, MapReduce, or other query capabilities?

Non-functional requirements

- **Performance and scalability**. What are your data performance requirements? Do you have specific requirements for data ingestion rates and data processing rates? What are the acceptable response times for querying and aggregation of data once ingested? How large will you need the data store to scale up? Is your workload more read-heavy or write-heavy?
- **Reliability**. What overall SLA do you need to support? What level of fault-tolerance do you need to provide for data consumers? What kind of backup and restore capabilities do you need?

- **Replication**. Will your data need to be distributed among multiple replicas or regions? What kind of data replication capabilities do you require?
- **Limits**. Will the limits of a particular data store support your requirements for scale, number of connections, and throughput?

Management and cost

- **Managed service**. When possible, use a managed data service, unless you require specific capabilities that can only be found in an IaaS-hosted data store.
- **Region availability**. For managed services, is the service available in all Azure regions? Does your solution need to be hosted in certain Azure regions?
- **Portability**. Will your data need to be migrated to on-premises, external datacenters, or other cloud hosting environments?
- **Licensing**. Do you have a preference of a proprietary versus OSS license type? Are there any other external restrictions on what type of license you can use?
- **Overall cost**. What is the overall cost of using the service within your solution? How many instances will need to run, to support your uptime and throughput requirements? Consider operations costs in this calculation. One reason to prefer managed services is the reduced operational cost.
- **Cost effectiveness**. Can you partition your data, to store it more cost effectively? For example, can you move large objects out of an expensive relational database into an object store?

Security

- **Security**. What type of encryption do you require? Do you need encryption at rest? What authentication mechanism do you want to use to connect to your data?
- **Auditing**. What kind of audit log do you need to generate?
- **Networking requirements**. Do you need to restrict or otherwise manage access to your data from other network resources? Does data need to be accessible only from inside the Azure environment? Does the data need to be accessible from specific IP addresses or subnets? Does it need to be accessible from applications or services hosted on-premises or in other external datacenters?

DevOps

- **Skill set**. Are there particular programming languages, operating systems, or other technology that your team is particularly adept at using? Are there others that would be difficult for your team to work with?
- **Clients** Is there good client support for your development languages?

Choose an analytical data store in Azure

11/19/2021 • 5 minutes to read • Edit Online

In a big data architecture, there is often a need for an analytical data store that serves processed data in a structured format that can be queried using analytical tools. Analytical data stores that support querying of both hot-path and cold-path data are collectively referred to as the serving layer, or data serving storage.

The serving layer deals with processed data from both the hot path and cold path. In the lambda architecture, the serving layer is subdivided into a *speed serving* layer, which stores data that has been processed incrementally, and a *batch serving* layer, which contains the batch-processed output. The serving layer requires strong support for random reads with low latency. Data storage for the speed layer should also support random writes, because batch loading data into this store would introduce undesired delays. On the other hand, data storage for the batch layer does not need to support random writes, but batch writes instead.

There is no single best data management choice for all data storage tasks. Different data management solutions are optimized for different tasks. Most real-world cloud apps and big data processes have a variety of data storage requirements and often use a combination of data storage solutions.

What are your options when choosing an analytical data store?

There are several options for data serving storage in Azure, depending on your needs:

- Azure Synapse Analytics
- Azure Synapse Spark pools
- Azure Databricks
- Azure Data Explorer
- Azure SQL Database
- SQL Server in Azure VM
- HBase/Phoenix on HDInsight
- Hive LLAP on HDInsight
- Azure Analysis Services
- Azure Cosmos DB

These options provide various database models that are optimized for different types of tasks:

- Key/value databases hold a single serialized object for each key value. They're good for storing large volumes of data where you want to get one item for a given key value and you don't have to query based on other properties of the item.
- Document databases are key/value databases in which the values are *documents*. A "document" in this context is a collection of named fields and values. The database typically stores the data in a format such as XML, YAML, JSON, or BSON, but may use plain text. Document databases can query on non-key fields and define secondary indexes to make querying more efficient. This makes a document database more suitable for applications that need to retrieve data based on criteria more complex than the value of the document key. For example, you could query on fields such as product ID, customer ID, or customer name.
- Column-family databases are key/value data stores that structure data storage into collections of related columns called column families. For example, a census database might have one group of columns for a person's name (first, middle, last), one group for the person's address, and one group for the person's profile information (data of birth, gender). The database can store each column family in a separate partition, while keeping all of the data for one person related to the same key. An application can read a single column family

without reading through all of the data for an entity.

- Graph databases store information as a collection of objects and relationships. A graph database can efficiently perform queries that traverse the network of objects and the relationships between them. For example, the objects might be employees in a human resources database, and you might want to facilitate queries such as "find all employees who directly or indirectly work for Scott."
- Telemetry and time series databases are an append-only collection of objects. Telemetry databases efficiently index data in a variety of column stores and in-memory structures, making them the optimal choice for storing and analyzing vast quantities of telemetry and time series data.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need serving storage that can serve as a hot path for your data? If yes, narrow your options to those that are optimized for a speed serving layer.
- Do you need massively parallel processing (MPP) support, where queries are automatically distributed across several processes or nodes? If yes, select an option that supports query scale out.
- Do you prefer to use a relational data store? If so, narrow your options to those with a relational database model. However, note that some non-relational stores support SQL syntax for querying, and tools such as PolyBase can be used to query non-relational data stores.
- Do you collect time series data? Do you use append-only data?

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Is managed service	Yes	Yes	Yes	Yes	Yes [1]	Yes [1]	Yes	Yes
Primary database model	Relational (columnar format when using columnstore indexes)	Relational tables with columnar storage	Wide column store	Relational (column store), telemetry, and time series store	Wide column store	Hive/In-Memory	Tabular semantic models	Document store, graph, key-value store, wide column store
SQL language support	Yes	Yes	Yes	Yes	Yes (using Phoenix JDBC driver)	Yes	No	Yes

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Optimized for speed serving layer	Yes [2]	Yes [3]	Yes	Yes	Yes	Yes	No	Yes

[1] With manual configuration and scaling.

[2] Using memory-optimized tables and hash or nonclustered indexes.

[3] Supported as an Azure Stream Analytics output.

Scalability capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Redundant regional servers for high availability	Yes	No	No	Yes	Yes	No	No	Yes
Supports query scale out	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic scalability (scale up)	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Supports in-memory caching of data	Yes	Yes	Yes	Yes	No	Yes	Yes	No

Security capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Authentication	SQL / Azure Active Directory (Azure AD)	SQL / Azure AD	Azure AD	local / Azure AD [1]	local / Azure AD [1]	Azure AD	database users / Azure AD via access control (IAM)

CAPABILITY	SQL DATABASE	AZURE SYNAPSE	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Data encryption at rest	Yes [2]	Yes [2]	Yes	Yes [1]	Yes [1]	Yes	Yes
Row-level security	Yes	Yes [3]	No	Yes [1]	Yes [1]	Yes	No
Supports firewalls	Yes	Yes	Yes	Yes [4]	Yes [4]	Yes	Yes
Dynamic data masking	Yes	Yes	Yes	Yes [1]	Yes	No	No

[1] Requires using a domain-joined HDInsight cluster.

[2] Requires using transparent data encryption (TDE) to encrypt and decrypt your data at rest.

[3] Filter predicates only. See Row-Level Security

[4] When used within an Azure Virtual Network. See Extend Azure HDInsight using an Azure Virtual Network.

Choose a data analytics technology in Azure

11/19/2021 • 4 minutes to read • Edit Online

The goal of most big data solutions is to provide insights into the data through analysis and reporting. This can include preconfigured reports and visualizations, or interactive data exploration.

What are your options when choosing a data analytics technology?

There are several options for analysis, visualizations, and reporting in Azure, depending on your needs:

- Power BI
- Jupyter Notebooks
- Zeppelin Notebooks
- Microsoft Azure Notebooks

Power BI

Power BI is a suite of business analytics tools. It can connect to hundreds of data sources, and can be used for ad hoc analysis. See this list of the currently available data sources. Use Power BI Embedded to integrate Power BI within your own applications without requiring any additional licensing.

Organizations can use Power BI to produce reports and publish them to the organization. Everyone can create personalized dashboards, with governance and security built in. Power BI uses Azure Active Directory (Azure AD) to authenticate users who log in to the Power BI service, and uses the Power BI login credentials whenever a user attempts to access resources that require authentication.

Jupyter Notebooks

Jupyter Notebooks provide a browser-based shell that lets data scientists create *notebook* files that contain Python, Scala, or R code and markdown text, making it an effective way to collaborate by sharing and documenting code and results in a single document.

Most varieties of HDInsight clusters, such as Spark or Hadoop, come preconfigured with Jupyter notebooks for interacting with data and submitting jobs for processing. Depending on the type of HDInsight cluster you are using, one or more kernels will be provided for interpreting and running your code. For example, Spark clusters on HDInsight provide Spark-related kernels that you can select from to execute Python or Scala code using the Spark engine.

Jupyter notebooks provide a great environment for analyzing, visualizing, and processing your data prior to building more advanced visualizations with a BI/reporting tool like Power BI.

Zeppelin Notebooks

Zeppelin Notebooks are another option for a browser-based shell, similar to Jupyter in functionality. Some HDInsight clusters come preconfigured with Zeppelin notebooks. However, if you are using an HDInsight Interactive Query (Hive LLAP) cluster, Zeppelin is currently your only choice of notebook that you can use to run interactive Hive queries. Also, if you are using a domain-joined HDInsight cluster, Zeppelin notebooks are the only type that enables you to assign different user logins to control access to notebooks and the underlying Hive tables.

Microsoft Azure Notebooks

Azure Notebooks is an online Jupyter Notebooks-based service that enables data scientists to create, run, and share Jupyter Notebooks in cloud-based libraries. Azure Notebooks provides execution environments for Python 2, Python 3, F#, and R, and provides several charting libraries for visualizing your data, such as ggplot,

matplotlib, bokeh, and seaborn.

Unlike Jupyter notebooks running on an HDInsight cluster, which are connected to the cluster's default storage account, Azure Notebooks does not provide any data. You must load data in a variety of ways, such downloading data from an online source, interacting with Azure Blobs or Table Storage, connecting to a SQL database, or loading data with the Copy Wizard for Azure Data Factory.

Key benefits:

- Free service—no Azure subscription required.
- No need to install Jupyter and the supporting R or Python distributions locally—just use a browser.
- Manage your own online libraries and access them from any device.
- Share your notebooks with collaborators.

Considerations:

- You will be unable to access your notebooks when offline.
- Limited processing capabilities of the free notebook service may not be enough to train large or complex models.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need to connect to numerous data sources, providing a centralized place to create reports for data spread throughout your domain? If so, choose an option that allows you to connect to 100s of data sources.
- Do you want to embed dynamic visualizations in an external website or application? If so, choose an option that provides embedding capabilities.
- Do you want to design your visualizations and reports while offline? If yes, choose an option with offline capabilities.
- Do you need heavy processing power to train large or complex AI models or work with very large data sets? If yes, choose an option that can connect to a big data cluster.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	POWER BI	JUPYTER NOTEBOOKS	ZEPPELIN NOTEBOOKS	MICROSOFT AZURE NOTEBOOKS
Connect to big data cluster for advanced processing	Yes	Yes	Yes	No
Managed service	Yes	Yes [1]	Yes [1]	Yes
Connect to 100s of data sources	Yes	No	No	No
Offline capabilities	Yes [2]	No	No	No

CAPABILITY	POWER BI	JUPYTER NOTEBOOKS	ZEPPELIN NOTEBOOKS	MICROSOFT AZURE NOTEBOOKS
Embedding capabilities	Yes	No	No	No
Automatic data refresh	Yes	No	No	No
Access to numerous open source packages	No	Yes [3]	Yes [3]	Yes [4]
Data transformation/cleansing options	Power Query, R	40 languages, including Python, R, Julia, and Scala	20+ interpreters, including Python, JDBC, and R	Python, F#, R
Pricing	Free for Power BI Desktop (authoring), see pricing for hosting options	Free	Free	Free
Multiuser collaboration	Yes	Yes (through sharing or with a multiuser server like JupyterHub)	Yes	Yes (through sharing)

[1] When used as part of a managed HDInsight cluster.

[2] With the use of Power BI Desktop.

[2] You can search the Maven repository for community-contributed packages.

[3] Python packages can be installed using either pip or conda. R packages can be installed from CRAN or GitHub. Packages in F# can be installed via nuget.org using the Paket dependency manager.

Choose a Microsoft cognitive services technology

11/19/2021 • 3 minutes to read • Edit Online

Microsoft cognitive services are cloud-based APIs that you can use in artificial intelligence (AI) applications and data flows. They provide you with pretrained models that are ready to use in your application, requiring no data and no model training on your part. The cognitive services are developed by Microsoft's AI and Research team and leverage the latest deep learning algorithms. They are consumed over HTTP REST interfaces. In addition, SDKs are available for many common application development frameworks.

The cognitive services include:

- Text analysis
- Computer vision
- Video analytics
- Speech recognition and generation
- Natural language understanding
- Intelligent search

Key benefits:

- Minimal development effort for state-of-the-art AI services.
- Easy integration into apps via HTTP REST interfaces.
- Built-in support for consuming cognitive services in Azure Data Lake Analytics.

Considerations:

- Only available over the web. Internet connectivity is generally required. An exception is the Custom Vision Service, whose trained model you can export for prediction on devices and at the IoT edge.
- Although considerable customization is supported, the available services may not suit all predictive analytics requirements.

What are your options when choosing amongst the cognitive services?

In Azure, there are dozens of Cognitive Services available. The current listing of these is available in a directory categorized by the functional area they support:

- Vision
- Speech
- Decision
- Search
- Language

Key selection criteria

To narrow the choices, start by answering these questions:

- What type of data are you dealing with? Narrow your options based on the type of input data you are working with. For example, if your input is text, select from the services that have an input type of text.

- Do you have the data to train a model? If yes, consider the custom services that enable you to train their underlying models with data that you provide, for improved accuracy and performance.

Capability matrix

The following tables summarize the key differences in capabilities.

Uses prebuilt models

CAPABILITY	INPUT TYPE	KEY BENEFIT
Text Analytics API	Text	Evaluate sentiment and topics to understand what users want.
Entity Linking API	Text	Power your app's data links with named entity recognition and disambiguation.
Language Understanding Intelligent Service (LUIS)	Text	Teach your apps to understand commands from your users.
QnA Maker Service	Text	Distill FAQ formatted information into conversational, easy-to-navigate answers.
Linguistic Analysis API	Text	Simplify complex language concepts and parse text.
Knowledge Exploration Service	Text	Enable interactive search experiences over structured data via natural language inputs.
Web Language Model API	Text	Use predictive language models trained on web-scale data.
Academic Knowledge API	Text	Tap into the wealth of academic content in the Microsoft Academic Graph populated by Bing.
Bing Autosuggest API	Text	Give your app intelligent autosuggest options for searches.
Bing Spell Check API	Text	Detect and correct spelling mistakes in your app.
Translator Text API	Text	Machine translation.
Recommendations API	Text	Predict and recommend items your customers want.
Bing Entity Search API	Text (web search query)	Identify and augment entity information from the web.
Bing Image Search API	Text (web search query)	Search for images.
Bing News Search API	Text (web search query)	Search for news.

CAPABILITY	INPUT TYPE	KEY BENEFIT
Bing Video Search API	Text (web search query)	Search for videos.
Bing Web Search API	Text (web search query)	Get enhanced search details from billions of web documents.
Bing Speech API	Text or Speech	Convert speech to text and back again.
Speaker Recognition API	Speech	Use speech to identify and authenticate individual speakers.
Translator Speech API	Speech	Perform real-time speech translation.
Computer Vision API	Images (or frames from video)	Distill actionable information from images, automatically create description of photos, derive tags, recognize celebrities, extract text, and create accurate thumbnails.
Content Moderator	Text, Images or Video	Automated image, text, and video moderation.
Emotion API	Images (photos with human subjects)	Identify the range emotions of human subjects.
Face API	Images (photos with human subjects)	Detect, identify, analyze, organize, and tag faces in photos.
Video Indexer	Video	Video insights such as sentiment, transcript speech, translate speech, recognize faces and emotions, and extract keywords.

Trained with custom data you provide

CAPABILITY	INPUT TYPE	KEY BENEFIT
Custom Vision Service	Images (or frames from video)	Customize your own computer vision models.
Custom Speech Service	Speech	Overcome speech recognition barriers like speaking style, background noise, and vocabulary.
Custom Decision Service	Web content (for example, RSS feed)	Use machine learning to automatically select the appropriate content for your home page
Bing Custom Search API	Text (web search query)	Commercial-grade search tool.

Compare the machine learning products and technologies from Microsoft

11/19/2021 • 8 minutes to read • Edit Online

Learn about the machine learning products and technologies from Microsoft. Compare options to help you choose how to most effectively build, deploy, and manage your machine learning solutions.

Cloud-based machine learning products

The following options are available for machine learning in the Azure cloud.

CLOUD OPTIONS	WHAT IT IS	WHAT YOU CAN DO WITH IT
Azure Machine Learning	Managed platform for machine learning	Use a pretrained model. Or, train, deploy, and manage models on Azure using Python and CLI
Azure Cognitive Services	Pre-built AI capabilities implemented through REST APIs and SDKs	Build intelligent applications quickly using standard programming languages. Doesn't require machine learning and data science expertise
Azure SQL Managed Instance Machine Learning Services	In-database machine learning for SQL	Train and deploy models inside Azure SQL Managed Instance
Machine learning in Azure Synapse Analytics	Analytics service with machine learning	Train and deploy models inside Azure SQL Managed Instance
Machine learning and AI with ONNX in Azure SQL Edge	Machine learning in SQL on IoT	Train and deploy models inside Azure SQL Edge
Azure Databricks	Apache Spark-based analytics platform	Build and deploy models and data workflows using integrations with open-source machine learning libraries and the MLFlow platform.

On-premises machine learning products

The following options are available for machine learning on-premises. On-premises servers can also run in a virtual machine in the cloud.

ON-PREMISES OPTIONS	WHAT IT IS	WHAT YOU CAN DO WITH IT
SQL Server Machine Learning Services	In-database machine learning for SQL	Train and deploy models inside SQL Server
Machine Learning Services on SQL Server Big Data Clusters	Machine learning in Big Data Clusters	Train and deploy models on SQL Server Big Data Clusters

Development platforms and tools

The following development platforms and tools are available for machine learning.

PLATFORMS/TOOLS	WHAT IT IS	WHAT YOU CAN DO WITH IT
Azure Data Science Virtual Machine	Virtual machine with pre-installed data science tools	Develop machine learning solutions in a pre-configured environment
ML.NET	Open-source, cross-platform machine learning SDK	Develop machine learning solutions for .NET applications
Windows ML	Windows 10 machine learning platform	Evaluate trained models on a Windows 10 device
MMLSpark	Open-source, distributed, machine learning and microservices framework for Apache Spark	Create and deploy scalable machine learning applications for Scala and Python.
Machine Learning extension for Azure Data Studio	Open-source and cross-platform machine learning extension for Azure Data Studio	Manage packages, import machine learning models, make predictions, and create notebooks to run experiments for your SQL databases

Azure Machine Learning

Azure Machine Learning is a fully managed cloud service used to train, deploy, and manage machine learning models at scale. It fully supports open-source technologies, so you can use tens of thousands of open-source Python packages such as TensorFlow, PyTorch, and scikit-learn. Rich tools are also available, such as Compute instances, Jupyter notebooks, or the Azure Machine Learning for Visual Studio Code extension, a free extension that allows you to manage your resources, model training workflows and deployments in Visual Studio Code. Azure Machine Learning includes features that automate model generation and tuning with ease, efficiency, and accuracy.

Use Python SDK, Jupyter notebooks, R, and the CLI for machine learning at cloud scale. For a low-code or no-code option, use Azure Machine Learning's interactive designer in the studio to easily and quickly build, test, and deploy models using pre-built machine learning algorithms.

Try Azure Machine Learning for free.

Type	Cloud-based machine learning solution
Supported languages	Python, R
Machine learning phases	Model training Deployment MLOps/Management
Key benefits	Code first (SDK) and studio & drag-and-drop designer web interface authoring options. Central management of scripts and run history, making it easy to compare model versions. Easy deployment and management of models to the cloud or edge devices.

Considerations	Requires some familiarity with the model management model.

Azure Cognitive Services

Azure Cognitive Services is a set of *pre-built* APIs that enable you to build apps that use natural methods of communication. The term pre-built suggests that you do not need to bring datasets or data science expertise to train models to use in your applications. That's all done for you and packaged as APIs and SDKs that allow your apps to see, hear, speak, understand, and interpret user needs with just a few lines of code. You can easily add intelligent features to your apps, such as:

- **Vision:** Object detection, face recognition, OCR, etc. See Computer Vision, Face, Form Recognizer.
- **Speech:** Speech-to-text, text-to-speech, speaker recognition, etc. See Speech Service.
- **Language:** Translation, Sentiment analysis, key phrase extraction, language understanding, etc. See Translator, Text Analytics, Language Understanding, QnA Maker
- **Decision:** Anomaly detection, content moderation, reinforcement learning. See Anomaly Detector, Content Moderator, Personalizer.

Use Cognitive Services to develop apps across devices and platforms. The APIs keep improving, and are easy to set up.

Type	APIs for building intelligent applications
Supported languages	Various options depending on the service. Standard ones are C#, Java, JavaScript, and Python.
Machine learning phases	Deployment
Key benefits	Build intelligent applications using pre-trained models available through REST API and SDK. Variety of models for natural communication methods with vision, speech, language, and decision. No machine learning or data science expertise required.

SQL machine learning

SQL machine learning adds statistical analysis, data visualization, and predictive analytics in Python and R for relational data, both on-premises and in the cloud. Current platforms and tools include:

- SQL Server Machine Learning Services
- Machine Learning Services on SQL Server Big Data Clusters
- Azure SQL Managed Instance Machine Learning Services
- Machine learning in Azure Synapse Analytics
- Machine learning and AI with ONNX in Azure SQL Edge
- Machine Learning extension for Azure Data Studio

Use SQL machine learning when you need built-in AI and predictive analytics on relational data in SQL.

Type	On-premises predictive analytics for relational data

Supported languages	Python, R, SQL
Machine learning phases	Data preparation Model training Deployment
Key benefits	Encapsulate predictive logic in a database function, making it easy to include in data-tier logic.
Considerations	Assumes a SQL database as the data tier for your application.

Azure Data Science Virtual Machine

The Azure Data Science Virtual Machine is a customized virtual machine environment on the Microsoft Azure cloud. It is available in versions for both Windows and Linux Ubuntu. The environment is built specifically for doing data science and developing ML solutions. It has many popular data science, ML frameworks, and other tools pre-installed and pre-configured to jump-start building intelligent applications for advanced analytics.

Use the Data Science VM when you need to run or host your jobs on a single node. Or if you need to remotely scale up your processing on a single machine.

Type	Customized virtual machine environment for data science
Key benefits	Reduced time to install, manage, and troubleshoot data science tools and frameworks. The latest versions of all commonly used tools and frameworks are included. Virtual machine options include highly scalable images with GPU capabilities for intensive data modeling.
Considerations	The virtual machine cannot be accessed when offline. Running a virtual machine incurs Azure charges, so you must be careful to have it running only when required.

Azure Databricks

Azure Databricks is an Apache Spark-based analytics platform optimized for the Microsoft Azure cloud services platform. Databricks is integrated with Azure to provide one-click setup, streamlined workflows, and an interactive workspace that enables collaboration between data scientists, data engineers, and business analysts. Use Python, R, Scala, and SQL code in web-based notebooks to query, visualize, and model data.

Use Databricks when you want to collaborate on building machine learning solutions on Apache Spark.

Type	Apache Spark-based analytics platform
Supported languages	Python, R, Scala, SQL

Machine learning phases	Data preparation Data preprocessing Model training Model tuning Model inference Management Deployment

ML.NET

ML.NET is an open-source, and cross-platform machine learning framework. With ML.NET, you can build custom machine learning solutions and integrate them into your .NET applications. ML.NET offers varying levels of interoperability with popular frameworks like TensorFlow and ONNX for training and scoring machine learning and deep learning models. For resource-intensive tasks like training image classification models, you can take advantage of Azure to train your models in the cloud.

Use ML.NET when you want to integrate machine learning solutions into your .NET applications. Choose between the API for a code-first experience and Model Builder or the CLI for a low-code experience.

Type	Open-source cross-platform framework for developing custom machine learning applications with .NET
Languages supported	C#, F#
Machine learning phases	Data preparation Training Deployment
Key benefits	Data science & ML experience not required Use familiar tools (Visual Studio, VS Code) and languages Deploy where .NET runs Extensible Scalable Local-first experience

Windows ML

Windows ML inference engine allows you to use trained machine learning models in your applications, evaluating trained models locally on Windows 10 devices.

Use Windows ML when you want to use trained machine learning models within your Windows applications.

Type	Inference engine for trained models in Windows devices
Languages supported	C#/C++, JavaScript

MMLSpark

Microsoft ML for Apache Spark (MMLSpark) is an open-source library that expands the distributed computing framework Apache Spark. MMLSpark adds many deep learning and data science tools to the Spark ecosystem, including seamless integration of Spark Machine Learning pipelines with Microsoft Cognitive Toolkit (CNTK),

LightGBM, LIME (Model Interpretability), and OpenCV. You can use these tools to create powerful predictive models on any Spark cluster, such as Azure Databricks or Cosmic Spark.

MMLSpark also brings new networking capabilities to the Spark ecosystem. With the HTTP on Spark project, users can embed any web service into their SparkML models. Additionally, MMLSpark provides easy-to-use tools for orchestrating Azure Cognitive Services at scale. For production-grade deployment, the Spark Serving project enables high throughput, submillisecond latency web services, backed by your Spark cluster.

Type	Open-source, distributed machine learning and microservices framework for Apache Spark
Languages supported	Scala 2.11, Java, Python 3.5+, R (beta)
Machine learning phases	Data preparation Model training Deployment
Key benefits	Scalability Streaming + Serving compatible Fault-tolerance
Considerations	Requires Apache Spark

Next steps

- To learn about all the Artificial Intelligence (AI) development products available from Microsoft, see Microsoft AI platform.
- For training in developing AI and Machine Learning solutions with Microsoft, see Microsoft Learn.

Choose a natural language processing technology in Azure

11/19/2021 • 3 minutes to read • Edit Online

Natural language processing (NLP) is used for tasks such as sentiment analysis, topic detection, language detection, key phrase extraction, and document categorization.

NLP can be use to classify documents, such as labeling documents as sensitive or spam. The output of NLP can be used for subsequent processing or search. Another use for NLP is to summarize text by identifying the entities present in the document. These entities can also be used to tag documents with keywords, which enables search and retrieval based on content. Entities might be combined into topics, with summaries that describe the important topics present in each document. The detected topics may be used to categorize the documents for navigation, or to enumerate related documents given a selected topic. Another use for NLP is to score text for sentiment, to assess the positive or negative tone of a document. These approaches use many techniques from natural language processing, such as:

- **Tokenizer**. Splitting the text into words or phrases.
- **Stemming and lemmatization**. Normalizing words so that different forms map to the canonical word with the same meaning. For example, "running" and "ran" map to "run."
- **Entity extraction**. Identifying subjects in the text.
- **Part of speech detection**. Identifying text as a verb, noun, participle, verb phrase, and so on.
- **Sentence boundary detection**. Detecting complete sentences within paragraphs of text.

When using NLP to extract information and insight from free-form text, the starting point is typically the raw documents stored in object storage such as Azure Storage or Azure Data Lake Store.

Challenges

- Processing a collection of free-form text documents is typically computationally resource intensive, as well as being time intensive.
- Without a standardized document format, it can be difficult to achieve consistently accurate results using free-form text processing to extract specific facts from a document. For example, think of a text representation of an invoice—it can be difficult to build a process that correctly extracts the invoice number and invoice date for invoices across any number of vendors.

What are your options when choosing an NLP service?

In Azure, the following services provide natural language processing (NLP) capabilities:

- Azure HDInsight with Spark and Spark MLlib
- Azure Databricks
- Microsoft Cognitive Services

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want to use prebuilt models? If yes, consider using the APIs offered by Microsoft Cognitive Services.
- Do you need to train custom models against a large corpus of text data? If yes, consider using Azure HDInsight with Spark MLlib and Spark NLP.
- Do you need low-level NLP capabilities like tokenization, stemming, lemmatization, and term frequency/inverse document frequency (TF/IDF)? If yes, consider using Azure HDInsight with Spark MLlib and Spark NLP.
- Do you need simple, high-level NLP capabilities like entity and intent identification, topic detection, spell check, or sentiment analysis? If yes, consider using the APIs offered by Microsoft Cognitive Services.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Provides pretrained models as a service	No	Yes
REST API	Yes	Yes
Programmability	Python, Scala, Java	C#, Java, Node.js, Python, PHP, Ruby
Support processing of big data sets and large documents	Yes	No

Low-level natural language processing capabilities

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Tokenizer	Yes (Spark NLP)	Yes (Linguistic Analysis API)
Stemmer	Yes (Spark NLP)	No
Lemmatizer	Yes (Spark NLP)	No
Part of speech tagging	Yes (Spark NLP)	Yes (Linguistic Analysis API)

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Term frequency/inverse-document frequency (TF/IDF)	Yes (Spark MLlib)	No
String similarity—edit distance calculation	Yes (Spark MLlib)	No
N-gram calculation	Yes (Spark MLlib)	No
Stop word removal	Yes (Spark MLlib)	No

High-level natural language processing capabilities

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Entity/intent identification and extraction	No	Yes (Language Understanding Intelligent Service (LUIS) API)
Topic detection	Yes (Spark NLP)	Yes (Text Analytics API)
Spell checking	Yes (Spark NLP)	Yes (Bing Spell Check API)
Sentiment analysis	Yes (Spark NLP)	Yes (Text Analytics API)
Language detection	No	Yes (Text Analytics API)
Supports multiple languages besides English	No	Yes (varies by API)

See also

Natural language processing

Understand Azure load balancing

11/19/2021 • 7 minutes to read • Edit Online

The term *load balancing* refers to the distribution of workloads across multiple computing resources. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overloading any single resource. It can also improve availability by sharing a workload across redundant computing resources.

Azure provides various load balancing services that you can use to distribute your workloads across multiple computing resources - Application Gateway, Front Door, Load Balancer, and Traffic Manager.

This article describes how you can use the Azure Load Balancing hub page in the Azure portal to determine an appropriate load-balancing solution for your business needs.

Service categorizations

Azure load balancing services can be categorized along two dimensions: global versus regional, and HTTP(S) versus non-HTTP(S).

Global versus regional

- **Global** load-balancing services distribute traffic across regional backends, clouds, or hybrid on-premises services. These services route end-user traffic to the closest available backend. They also react to changes in service reliability or performance, in order to maximize availability and performance. You can think of them as systems that load balance between application stamps, endpoints, or scale-units hosted across different regions/geographies.
- **Regional** load-balancing services distribute traffic within virtual networks across virtual machines (VMs) or zonal and zone-redundant service endpoints within a region. You can think of them as systems that load balance between VMs, containers, or clusters within a region in a virtual network.

HTTP(S) versus non-HTTP(S)

- **HTTP(S)** load-balancing services are Layer 7 load balancers that only accept HTTP(S) traffic. They are intended for web applications or other HTTP(S) endpoints. They include features such as SSL offload, web application firewall, path-based load balancing, and session affinity.
- **Non-HTTP/S** load-balancing services can handle non-HTTP(S) traffic and are recommended for non-web workloads.

The following table summarizes the Azure load balancing services by these categories:

SERVICE	GLOBAL/REGIONAL	RECOMMENDED TRAFFIC
Azure Front Door	Global	HTTP(S)
Traffic Manager	Global	non-HTTP(S)
Application Gateway	Regional	HTTP(S)
Azure Load Balancer	Regional	non-HTTP(S)

Azure load balancing services

Here are the main load-balancing services currently available in Azure:

Front Door is an application delivery network that provides global load balancing and site acceleration service for web applications. It offers Layer 7 capabilities for your application like SSL offload, path-based routing, fast failover, caching, etc. to improve performance and high-availability of your applications.

> **NOTE**
>
> At this time, Azure Front Door does not support Web Sockets.

Traffic Manager is a DNS-based traffic load balancer that enables you to distribute traffic optimally to services across global Azure regions, while providing high availability and responsiveness. Because Traffic Manager is a DNS-based load-balancing service, it load balances only at the domain level. For that reason, it can't fail over as quickly as Front Door, because of common challenges around DNS caching and systems not honoring DNS TTLs.

Application Gateway provides application delivery controller (ADC) as a service, offering various Layer 7 load-balancing capabilities. Use it to optimize web farm productivity by offloading CPU-intensive SSL termination to the gateway.

Azure Load Balancer is a high-performance, ultra low-latency Layer 4 load-balancing service (inbound and outbound) for all UDP and TCP protocols. It is built to handle millions of requests per second while ensuring your solution is highly available. Azure Load Balancer is zone-redundant, ensuring high availability across Availability Zones.

Choose a load balancing solution using Azure portal

You can use the Azure Load Balancing page in the Azure portal to help you guide to the right load-balancing solution for your business need. Azure Load Balancing includes the decision making queries described in the workflow of the following section and can be accessed as follows:

1. Sign in to the Azure portal at https://portal.azure.com.
2. In the *Search resources, services, and docs* box at the top of the portal, type *load balancing*. When **Load balancing** appears in the search results, select it.

> **NOTE**
>
> To learn about some of the factors considered in selecting a load-balancing solution, see Decision tree for load-balancing in Azure in this article.

3. In the **Load balancing - help me choose (Preview)** page, do one of the following:

   - To find the appropriate load-balancing solution for your business, follow instructions in the default **Help me choose** tab.

   - To learn about the supported protocols and service capabilities of each load balancing service, select the **Service comparisons** tab.
   - To access free training on load balancing services, select the **Tutorial** tab.

Reference architecture examples

The following table lists various architecture reference articles based on the load-balancing services used as a solution.

SERVICE(S)	ARTICLE	DESCRIPTION

SERVICE(S)	ARTICLE	DESCRIPTION
Load Balancer	Load balance virtual machines (VMs) across availability zones	Load balance VMs across availability zones helps to protect your apps and data from an unlikely failure or loss of an entire datacenter. With zone-redundancy, one or more availability zones can fail and the data path survives as long as one zone in the region remains healthy.
Front Door	Sharing location in real time using low-cost serverless Azure services	Use Azure Front Door to provide higher availability for your applications than deploying to a single region. If a regional outage affects the primary region, you can use Front Door to fail over to the secondary region.
Application Gateway	IaaS: Web application with relational database	Learn how to use resources spread across multiple zones to provide a high availability (HA) architecture for hosting an Infrastructure as a Service (IaaS) web application and SQL Server database.
Traffic Manager	Multi-tier web application built for high availability and disaster recovery	Deploy resilient multi-tier applications built for high availability and disaster recovery. If the primary region becomes unavailable, Traffic Manager fails over to the secondary region.
Azure Front Door + Application Gateway	Multitenant SaaS on Azure	Use a multi-tenant solution that includes a combination of Front Door and Application Gateway. Front Door helps load balance traffic across regions and Application Gateway routes and load-balances traffic internally in the application to the various services that satisfy client business needs.
Traffic Manager + Load Balancer	Multi-region N-tier application	A multi-region N-tier application that uses Traffic Manager to route incoming requests to a primary region and if that region becomes unavailable, Traffic Manager fails over to the secondary region.
Traffic Manager + Application Gateway	Multi-region load balancing with Traffic Manager and Application Gateway	Learn how to serve web workloads and deploy resilient multi-tier applications in multiple Azure regions, in order to achieve high availability and a robust disaster recovery infrastructure.

Decision tree for load balancing in Azure

When selecting the load-balancing options, here are some factors that are considered when you select the **Help me choose** default tab in Azure load balancing:

- **Traffic type**. Is it a web (HTTP/HTTPS) application? Is it public facing or a private application?
- **Global versus. regional**. Do you need to load balance VMs or containers within a virtual network, or load balance scale unit/deployments across regions, or both?
- **Availability**. What is the service SLA?
- **Cost**. See Azure pricing. In addition to the cost of the service itself, consider the operations cost for managing a solution built on that service.
- **Features and limits**. What are the overall limitations of each service? See Service limits.

The following flowchart will help you to choose a load-balancing solution for your application. The flowchart guides you through a set of key decision criteria to reach a recommendation.

Treat this flowchart as a starting point. Every application has unique requirements, so use the recommendation as a starting point. Then perform a more detailed evaluation.

If your application consists of multiple workloads, evaluate each workload separately. A complete solution may incorporate two or more load-balancing solutions.

Definitions

- **Internet facing**. Applications that are publicly accessible from the internet. As a best practice, application owners apply restrictive access policies or protect the application by setting up offerings like web application firewall and DDoS protection.

- **Global**. End users or clients are located beyond a small geographical area. For example, users across multiple continents, across countries/regions within a continent, or even across multiple metropolitan areas within a larger country/region.
- **PaaS**. Platform as a service (PaaS) services provide a managed hosting environment, where you can deploy your application without needing to manage VMs or networking resources. In this case, PaaS refers to services that provide integrated load balancing within a region. See Choosing a compute service – Scalability.
- **AKS**. Azure Kubernetes Service enables you to deploy and manage containerized applications. AKS provides serverless Kubernetes, an integrated continuous integration and continuous delivery (CI/CD) experience, and enterprise-grade security and governance. For more information about our AKS architectural resources, see Azure Kubernetes Service (AKS) architecture design.
- **IaaS**. Infrastructure as a service (IaaS) is a computing option where you provision the VMs that you need, along with associated network and storage components. IaaS applications require internal load balancing within a virtual network, using Azure Load Balancer.
- **Application-layer processing** refers to special routing within a virtual network. For example, path-based routing within the virtual network across VMs or virtual machine scale sets. For more information, see When should we deploy an Application Gateway behind Front Door?.

Next steps

- Create a public load balancer to load balance VMs
- Direct web traffic with Application Gateway
- Configure Traffic Manager for global DNS-based load balancing
- Configure Front Door for a highly available global web application

Asynchronous messaging options in Azure

11/19/2021 • 17 minutes to read • Edit Online

This article describes the different types of messages and the entities that participate in a messaging infrastructure. Based on the requirements of each message type, the article recommends Azure messaging services. The options include Azure Service Bus, Event Grid, and Event Hubs.

At an architectural level, a message is a datagram created by an entity (*producer*), to distribute information so that other entities (*consumers*) can be aware and act accordingly. The producer and the consumer can communicate directly or optionally through an intermediary entity (*message broker*). This article focuses on asynchronous messaging using a message broker.

Messages can be classified into two main categories. If the producer expects an action from the consumer, that message is a *command*. If the message informs the consumer that an action has taken place, then the message is an *event*.

Commands

The producer sends a command with the intent that the consumer(s) will perform an operation within the scope of a business transaction.

A command is a high-value message and must be delivered at least once. If a command is lost, the entire business transaction might fail. Also, a command shouldn't be processed more than once. Doing so might cause an erroneous transaction. A customer might get duplicate orders or billed twice.

Commands are often used to manage the workflow of a multistep business transaction. Depending on the business logic, the producer may expect the consumer to acknowledge the message and report the results of the operation. Based on that result, the producer may choose an appropriate course of action.

Events

An event is a type of message that a producer raises to announce facts.

The producer (known as the *publisher* in this context) has no expectations that the events will result in any action.

Interested consumer(s), can subscribe, listen for events, and take actions depending on their consumption scenario. Events can have multiple subscribers or no subscribers at all. Two different subscribers can react to an event with different actions and not be aware of one another.

The producer and consumer are loosely coupled and managed independently. The consumer isn't expected to acknowledge the event back to the producer. A consumer that is no longer interested in the events, can unsubscribe. The consumer is removed from the pipeline without affecting the producer or the overall functionality of the system.

There are two categories of events:

- The producer raises events to announce discrete facts. A common use case is event notification. For example, Azure Resource Manager raises events when it creates, modifies, or deletes resources. A subscriber of those events could be a Logic App that sends alert emails.
- The producer raises related events in a sequence, or a stream of events, over a period of time. Typically, a stream is consumed for statistical evaluation. The evaluation can be done within a temporal window or as events arrive. Telemetry is a common use case, for example, health and load monitoring of a system. Another case is event streaming from IoT devices.

A common pattern for implementing event messaging is the Publisher-Subscriber pattern.

Role and benefits of a message broker

An intermediate message broker provides the functionality of moving messages from producer to consumer and can offer additional benefits.

Decoupling

A message broker decouples the producer from the consumer in the logic that generates and uses the messages, respectively. In a complex workflow, the broker can encourage business operations to be decoupled and help coordinate the workflow.

For example, a single business transaction requires distinct operations that are performed in a business logic sequence. The producer issues a command that signals a consumer to start an operation. The consumer acknowledges the message in a separate queue reserved for lining up responses for the producer. Only after receiving the response, the producer sends a new message to start the next operation in the sequence. A different consumer processes that message and sends a completion message to the response queue. By using messaging, the services coordinate the workflow of the transaction among themselves.

A message broker provides temporal decoupling. The producer and consumer don't have to run concurrently. A producer can send a message to the message broker regardless of the availability of the consumer. Conversely, the consumer isn't restricted by the producer's availability.

For example, the user interface of a web app generates messages and uses a queue as the message broker. When ready, consumers can retrieve messages from the queue and perform the work. Temporal decoupling helps the user interface to remain responsive. It's not blocked while the messages are handled asynchronously.

Certain operations can take long to complete. After issuing a command, the producer shouldn't have to wait until the consumer completes it. A message broker helps asynchronous processing of messages.

Load balancing

Producers may post a large number of messages that are serviced by many consumers. Use a message broker to distribute processing across servers and improve throughput. Consumers can run on different servers to spread the load. Consumers can be added dynamically to scale out the system when needed or removed otherwise.

The Competing Consumers Pattern explains how to process multiple messages concurrently to optimize throughput, to improve scalability and availability, and to balance the workload.

Load leveling

The volume of messages generated by the producer or a group of producers can be variable. At times there might be a large volume causing spikes in messages. Instead of adding consumers to handle this work, a

message broker can act as a buffer, and consumers gradually drain messages at their own pace without stressing the system.

The Queue-based Load Leveling Pattern provides more information.

Reliable messaging

A message broker helps ensure that messages aren't lost even if communication fails between the producer and consumer. The producer can post messages to the message broker and the consumer can retrieve them when communication is reestablished. The producer isn't blocked unless it loses connectivity with the message broker.

Resilient messaging

A message broker can add resiliency to the consumers in your system. If a consumer fails while processing a message, another instance of the consumer can process that message. The reprocessing is possible because the message persists in the broker.

Technology choices for a message broker

Azure provides several message broker services, each with a range of features. Before choosing a service, determine the intent and requirements of the message.

Azure Service Bus

Azure Service Bus queues are well suited for transferring commands from producers to consumers. Here are some considerations.

Pull model

A consumer of a Service Bus queue constantly polls Service Bus to check if new messages are available. The client SDKs and Azure Functions trigger for Service Bus abstract that model. When a new message is available, the consumer's callback is invoked, and the message is sent to the consumer.

Guaranteed delivery

Service Bus allows a consumer to peek the queue and lock a message from other consumers.

It's the responsibility of the consumer to report the processing status of the message. Only when the consumer marks the message as consumed, Service Bus removes the message from the queue. If a failure, timeout, or crash occurs, Service Bus unlocks the message so that other consumers can retrieve it. This way messages aren't lost in transfer.

A producer might accidentally send the same message twice. For instance, a producer instance fails after sending a message. Another producer replaces the original instance and sends the message again. Azure Service Bus queues provide a built-in de-duping capability that detects and removes duplicate messages. There's

still a chance that a message is delivered twice. For example, if a consumer fails while processing, the message is returned to the queue and is retrieved by the same or another consumer. The message processing logic in the consumer should be idempotent so that even if the work is repeated, the state of the system isn't changed.

Message ordering

If you want consumers to get the messages in the order they are sent, Service Bus queues guarantee first-in-first-out (FIFO) ordered delivery by using sessions. A session can have one or more messages. The messages are correlated with the **SessionId** property. Messages that are part of a session, never expire. A session can be locked to a consumer to prevent its messages from being handled by a different consumer.

For more information, see Message Sessions.

Message persistence

Service bus queues support temporal decoupling. Even when a consumer isn't available or unable to process the message, it remains in the queue.

Checkpoint long-running transactions

Business transactions can run for a long time. Each operation in the transaction can have multiple messages. Use checkpointing to coordinate the workflow and provide resiliency in case a transaction fails.

Service Bus queues allow checkpointing through the session state capability. State information is incrementally recorded in the queue (**SetState**) for messages that belong to a session. For example, a consumer can track progress by checking the state (**GetState**) every now and then. If a consumer fails, another consumer can use state information to determine the last known checkpoint to resume the session.

Dead-letter queue (DLQ)

A Service Bus queue has a default subqueue, called the dead-letter queue (DLQ) to hold messages that couldn't be delivered or processed. Service Bus or the message processing logic in the consumer can add messages to the DLQ. The DLQ keeps the messages until they are retrieved from the queue.

Here are examples when a message can end up being in the DLQ:

- A poison message is a message that cannot be handled because it's malformed or contains unexpected information. In Service Bus queues, you can detect poison messages by setting the **MaxDeliveryCount** property of the queue. If number of times the same message is received exceeds that property value, Service Bus moves the message to the DLQ.
- A message might no longer be relevant if it isn't processed within a period. Service Bus queues allow the producer to post messages with a time-to-live attribute. If this period expires before the message is received, the message is placed in the DLQ.

Examine messages in the DLQ to determine the reason for failure.

Hybrid solution

Service Bus bridges on-premises systems and cloud solutions. On-premises systems are often difficult to reach because of firewall restrictions. Both the producer and consumer (either can be on-premises or the cloud) can use the Service Bus queue endpoint as the pickup and drop off location for messages.

Topics and subscriptions

Service Bus supports the Publisher-Subscriber pattern through Service Bus topics and subscriptions.

This feature provides a way for the producer to broadcast messages to multiple consumers. When a topic receives a message, it's forwarded to all the subscribed consumers. Optionally, a subscription can have filter criteria that allows the consumer to get a subset of messages. Each consumer retrieves messages from a subscription in a similar way to a queue.

For more information, see Azure Service Bus topics.

Azure Event Grid

Azure Event Grid is recommended for discrete events. Event Grid follows the Publisher-Subscriber pattern. When event sources trigger events, they are published to Event grid topics. Consumers of those events create Event Grid subscriptions by specifying event types and event handler that will process the events. If there are no subscribers, the events are discarded. Each event can have multiple subscriptions.

Push Model

Event Grid propagates messages to the subscribers in a push model. Suppose you have an event grid subscription with a webhook. When a new event arrives, Event Grid posts the event to the webhook endpoint.

Integrated with Azure

Choose Event Grid if you want to get notifications about Azure resources. Many Azure services act as event sources that have built-in Event Grid topics. Event Grid also supports various Azure services that can be configured as event handlers. It's easy to subscribe to those topics to route events to event handlers of your choice. For example, you can use Event Grid to invoke an Azure Function when a blob storage is created or deleted.

Custom topics

Create custom Event Grid topics, if you want to send events from your application or an Azure service that isn't integrated with Event Grid.

For example, to see the progress of an entire business transaction, you want the participating services to raise events as they are processing their individual business operations. A web app shows those events. One way is to create a custom topic and add a subscription with your web app registered through an HTTP WebHook. As business services send events to the custom topic, Event Grid pushes them to your web app.

Filtered events

You can specify filters in a subscription to instruct Event Grid to route only a subset of events to a specific event handler. The filters are specified in the subscription schema. Any event sent to the topic with values that match the filter are automatically forwarded to that subscription.

For example, content in various formats are uploaded to Blob Storage. Each time a file is added, an event is raised and published to Event Grid. The event subscription might have a filter that only sends events for images so that an event handler can generate thumbnails.

For more information about filtering, see Filter events for Event Grid.

High throughput

Event Grid can route 10,000,000 events per second per region. The first 100,000 operations per month are free. For cost considerations, see How much does Event Grid cost?

Resilient delivery

Even though successful delivery for events isn't as crucial as commands, you might still want some guarantee depending on the type of event. Event Grid offers features that you can enable and customize, such as retry policies, expiration time, and dead lettering. For more information, see Delivery and retry.

Event Grid's retry process can help resiliency but it's not fail-safe. In the retry process, Event Grid might deliver the message more than once, skip, or delay some retries if the endpoint is unresponsive for a long time. For more information, see Retry schedule and duration.

You can persist undelivered events to a blob storage account by enabling dead-lettering. There's a delay in delivering the message to the blob storage endpoint and if that endpoint is unresponsive, then Event Grid discards the event. For more information, see Dead letter and retry policies.

Azure Event Hubs

When working with an event stream, Azure Event Hubs is the recommended message broker. Essentially, it's a large buffer that's capable of receiving large volumes of data with low latency. The received data can be read quickly through concurrent operations. You can transform the received data by using any real-time analytics provider. Event Hubs also provides the capability to store events in a storage account.

Fast ingestion

Event Hubs is capable of ingesting millions of events per second. The events are only appended to the stream and are ordered by time.

Pull model

Like Event Grid, Event Hubs also offers Publisher-Subscriber capabilities. A key difference between Event Grid and Event Hubs is in the way event data is made available to the subscribers. Event Grid pushes the ingested data to the subscribers whereas Event Hub makes the data available in a pull model. As events are received, Event Hubs appends them to the stream. A subscriber manages its cursor and can move forward and back in the stream, select a time offset, and replay a sequence at its pace.

Stream processors are subscribers that pull data from Event Hubs for the purposes of transformation and statistical analysis. Use Azure Stream Analytics and Apache Spark for complex processing such as aggregation over time windows or anomaly detection.

If you want to act on each event per partition, you can pull the data by using Event Processing Host or by using built in connector such as Logic Apps to provide the transformation logic. Another option is to use Azure Functions.

Partitioning

A partition is a portion of the event stream. The events are divided by using a partition key. For example, several IoT devices send device data to an event hub. The partition key is the device identifier. As events are ingested, Event Hubs moves them to separate partitions. Within each partition, all events are ordered by time.

A consumer is an instance of code that processes the event data. Event Hubs follows a partitioned consumer pattern. Each consumer only reads a specific partition. Having multiple partitions results in faster processing because the stream can be read concurrently by multiple consumers.

Instances of the same consumer make up a single consumer group. Multiple consumer groups can read the same stream with different intentions. Suppose an event stream has data from a temperature sensor. One consumer group can read the stream to detect anomalies such as a spike in temperature. Another can read the same stream to calculate a rolling average temperature in a temporal window.

Event Hubs supports the Publisher-Subscriber pattern by allowing multiple consumer groups. Each consumer group is a subscriber.

For more information about Event Hub partitioning, see Partitions.

Event Hubs Capture

The Capture feature allows you to store the event stream to an Azure Blob storage or Data Lake Storage. This way of storing events is reliable because even if the storage account isn't available, Capture keeps your data for a period, and then writes to the storage after it's available.

> Storage services can also offer additional features for analyzing events. For example, by taking advantage of the access tiers of a blob storage account, you can store events in a hot tier for data that needs frequent access. You might use that data for visualization. Alternately, you can store data in the archive tier and retrieve it occasionally for auditing purposes.

Capture stores *all* events ingested by Event Hubs and is useful for batch processing. You can generate reports on the data by using a MapReduce function. Captured data can also serve as the source of truth. If certain facts were missed while aggregating the data, you can refer to the captured data.

For details about this feature, see Capture events through Azure Event Hubs in Azure Blob Storage or Azure Data Lake Storage.

Support for Apache Kafka clients

Event Hubs provides an endpoint for Apache Kafka clients. Existing clients can update their configuration to

point to the endpoint and start sending events to Event Hubs. No code changes are required.

For more information, see Event Hubs for Apache Kafka.

Crossover scenarios

In some cases, it's advantageous to combine two messaging services.

Combining services can increase the efficiency of your messaging system. For instance, in your business transaction, you use Azure Service Bus queues to handle messages. Queues that are mostly idle and receive messages occasionally are inefficient because the consumer is constantly polling the queue for new messages. You can set up an Event Grid subscription with an Azure Function as the event handler. Each time the queue receives a message and there are no consumers listening, Event Grid sends a notification, which invokes the Azure Function that drains the queue.

For details about connecting Service Bus to Event Grid, see Azure Service Bus to Event Grid integration overview.

The Enterprise integration on Azure using message queues and events reference architecture shows an implementation of Service Bus to Event Grid integration.

Here's another example. Event Grid receives a set of events in which some events require a workflow while others are for notification. The message metadata indicates the type of event. One way is to check the metadata by using the filtering feature in the event subscription. If it requires a workflow, Event Grid sends it to Azure Service Bus queue. The receivers of that queue can take necessary actions. The notification events are sent to Logic Apps to send alert emails.

Related patterns

Consider these patterns when implementing asynchronous messaging:

- Competing Consumers Pattern. Multiple consumers may need to compete to read messages from a queue. This pattern explains how to process multiple messages concurrently to optimize throughput, to improve

scalability and availability, and to balance the workload.

- Priority Queue Pattern. For cases where the business logic requires that some messages are processed before others, this pattern describes how messages posted by a producer that have a higher priority can be received and processed more quickly by a consumer than messages of a lower priority.
- Queue-based Load Leveling Pattern. This pattern uses a message broker to act as a buffer between a producer and a consumer to help to minimize the impact on availability and responsiveness of intermittent heavy loads for both those entities.
- Retry Pattern. A producer or consumer might be unable connect to a queue, but the reasons for this failure may be temporary and quickly pass. This pattern describes how to handle this situation to add resiliency to an application.
- Scheduler Agent Supervisor Pattern. Messaging is often used as part of a workflow implementation. This pattern demonstrates how messaging can coordinate a set of actions across a distributed set of services and other remote resources, and enable a system to recover and retry actions that fail.
- Choreography pattern. This pattern shows how services can use messaging to control the workflow of a business transaction.
- Claim-Check Pattern. This pattern shows how to split a large message into a claim check and a payload.

Community resources

Jonathon Oliver's blog post: Idempotency

Martin Fowler's blog post: What do you mean by "Event-Driven"?

Compare Internet of Things (IoT) solution approaches (PaaS vs. aPaaS)

11/19/2021 • 7 minutes to read • Edit Online

IoT solutions require a combination of technologies to effectively connect devices, events, and actions to cloud applications. In Azure, we have a single set of guidance for building and connecting devices to the cloud. However, there are many options for building and deploying your IoT cloud solutions. Which technologies and services you'll use depends on your scenario's development, deployment, and management needs.

Starting with Azure IoT Central (aPaaS)

Using an application platform as a service (aPaaS) streamlines many of the complex decisions you'll face when building an IoT solution. Azure IoT Central is Microsoft's aPaaS offering designed to simplify and accelerate IoT solution assembly and operation by assembling platform as a service (PaaS) components into an extensible and fully managed app development platform hosted by Microsoft. This takes a lot of the guesswork and complexity out of building reliable, scalable, and secure IoT applications.

An out-of-the box web UX and API surface area make it simple to monitor device conditions, create rules, and manage millions of devices and their data remotely throughout their life cycles. Furthermore, it enables you to act on device insights by extending IoT intelligence into line-of-business applications. Azure IoT Central also offers built-in disaster recovery, multitenancy, global availability, and a predictable cost structure.

Building with Azure PaaS services

In certain scenarios you may need a higher degree of control and customization than Azure IoT Central provides. In these cases, Azure also offers individual platform as a service (PaaS) cloud services that you can use to build a custom IoT solution. For example, you might build a solution using a combination of these PaaS services:

- Azure IoT Device Provisioning Service and Azure IoT Hub for device provisioning, device connectivity, and management
- Azure Time Series Insights for storing and analyzing warm and cold path time series data from IoT devices

- Azure Stream Analytics for analyzing hot path data from IoT devices
- Azure IoT Edge for running AI, third-party services, or your own business logic on IoT Edge devices

Comparing approaches

Choosing to build with Azure IoT Central gives you the opportunity to focus time and money on transforming your business and designing innovative offerings, rather than maintaining and updating a complex and continually evolving IoT infrastructure. However, if your solution requires features or services that Azure IoT Central does not currently support, you may need to develop a PaaS solution using Azure IoT Hub as a core element.

You can use the table and links below to help decide if you can use a managed solution based on Azure IoT Central, or if you should consider building a PaaS solution using Azure IoT Hub.

	AZURE IOT CENTRAL	AZURE IOT HUB

	AZURE IOT CENTRAL	AZURE IOT HUB
Type of Service	Fully managed aPaaS solution. It simplifies device connectivity and management at scale so that you can focus time and resources on using IoT for business transformation. This simplicity comes with a tradeoff: an aPaaS-based solution is less customizable than a PaaS-based solution.	Managed PaaS back-end solution that acts as a central message hub between your IoT application and the devices it manages. You can build more functionality using additional Azure PaaS services. This approach provides great flexibility but requires more development and management effort to build and operate your solution.
Application Template	Application templates in Azure IoT Central help solution builders kick-start IoT solution development. You can get started with a generic application template, or use a prebuilt industry-focused application template for retail, energy, government, or healthcare.	Not supported. You'll design and build your own solution using Azure IoT Hub and other PaaS services.
Device Management	Provides seamless device integration and device management capability. Device Provisioning Service capabilities (DPS) are built in.	No built-in experience. You'll design and build your own solutions using Azure IoT Hub primitives, such as device twin and direct methods. DPS must be enabled separately.
Scalability	Supports auto-scaling.	There is no built-in mechanism for automatically scaling an IoT Hub. You'll need to deploy other solutions to enable auto-scaling. See: Auto-scale your Azure IoT Hub
Message Retention	Retains data on a rolling, 30-day basis. You can continuously export data using the export feature.	Allows data retention in the built-in Event Hubs for a maximum of 7 days.
Visualizations	Yes, IoT Central has a UX that makes it simple to visualize device data, perform analytics queries, and create custom dashboards.	No.
OPC UA Protocol	Not yet.	OPC Publisher is a Microsoft-supported open-source product that bridges the gap between industrial assets and Azure hosted resources. It connects to OPC UA–enabled assets or industrial connectivity software and publishes telemetry data to Azure IoT Hub in various formats, including IEC62541 OPC UA PubSub standard format. See: Azure/iot-edge-opc-publisher: Microsoft OPC Publisher

	AZURE IOT CENTRAL	AZURE IOT HUB
Pricing	The first two active devices within an IoT Central application are free, if their message volume does not exceed 800 (Standard Tier 0 plan), 10,000 (Standard Tier 1 plan), or 60,000 (Standard Tier 2 plan) per month. Volumes exceeding those thresholds will incur overage charges. Beyond that, device pricing is prorated monthly. For each hour during the billing period, the highest number of active devices is counted and billed. See: Azure IoT Central pricing	See: Azure IoT Hub pricing
Analytics, Insights, and Actions	Integrated analytics experience targeted at exploration of device data in the context of device management.	You'll use separate Azure PaaS services to incorporate analytics, insights, and actions, like Azure Steam Analytics, Time Series Insight, Azure Data Explorer, and Azure Synapse.
Big Data Management	Data Management can be managed from Azure IoT Central itself.	You'll need to add and manage big data Azure PaaS services as part of your solution.
High Availability and Disaster Recovery	High availability and disaster recovery capabilities are built in to Azure IoT Central and managed for you automatically. See: Best practices for device development in Azure IoT Central	Can be configured to support multiple high availability and disaster recovery scenarios. See: Azure IoT Hub high availability and disaster recovery
SLA	Azure IoT Central guarantees you 99.9% connectivity. See: SLA for Azure IoT Central	The Azure IoT Hub standard and basic tiers guarantee 99.9% uptime. No SLA is provided for the Free Tier of Azure IoT Hub. See: SLA for Azure IoT Hub
Device Template	Supports centrally defining and managing device templates that help structure the characteristics and behaviors of device types for use in supported device management tasks and visualizations.	Requires users to create their own repository to define and manage device message templates.
Data Export	Provides data export to Azure blob storage, event hubs, service bus, webhook, and Azure Data Explorer. Additional capabilities include filtering, enriching, and transforming messages on egress.	Provides a built-in event hub endpoint and can also make use of message routing to export data to other storage locations.

	AZURE IOT CENTRAL	AZURE IOT HUB
Multi-tenancy	IoT Central Organizations enabled in-app multi-tenancy where you to define a hierarchy to manage which users can see which devices in your IoT Central application.	Not supported. Tenancy can be achieved by using separate hubs per customer and/or access control can be built into the data layer of solutions.
Rules and Actions	Provides a built-in rules and actions processing capability with email notification, Azure Monitor group, Power Automate, and Webhook actions. See: What is Azure IoT Central?	Data coming from IoT Hub can be sent to Azure Stream Analytics, Azure Time Series Insights, or Azure Event Grid. From those services you can connect to Azure Logic apps or other custom applications to handle rules and actions processing. See: IoT remote monitoring and notifications with Azure Logic Apps
SigFox/LoRaWAN Protocol	Uses IoT Central Device Bridge. See: Azure IoT Central Device Bridge	Requires you to write a custom Module on Azure IoT Edge and integrate it with Azure IoT Hub.

Next steps

Continue learning about IoT Hub and IoT Central:

- What is Azure IoT Central?
- What is Azure IoT Hub?

Related resources

Additional IoT topics:

- Overview of device management with Azure IoT Hub
- Azure IoT Hub high availability and disaster recovery
- Understand and use Azure IoT Hub SDKs
- IoT remote monitoring and notifications with Azure Logic Apps

IoT architecture guides:

- IoT solutions conceptual overview
- Vision with Azure IoT Edge
- Azure Industrial IoT Analytics Guidance
- Azure IoT reference architecture
- IoT and data analytics

Example architectures using Azure IoT Central:

- Retail - Buy online, pickup in store (BOPIS)
- Environment monitoring and supply chain optimization with IoT
- Blockchain workflow application

Example architectures using Azure IoT Hub:

- Azure IoT reference architecture
- IoT and data analytics
- IoT using Cosmos DB
- Predictive maintenance with the intelligent IoT Edge
- Predictive Maintenance for Industrial IoT
- Project 15 Open Platform
- IoT connected light, power, and internet for emerging markets
- Condition Monitoring for Industrial IoT

Best practices in cloud applications

11/19/2021 • 2 minutes to read • Edit Online

These best practices can help you build reliable, scalable, and secure applications in the cloud. They offer guidelines and tips for designing and implementing efficient and robust systems, mechanisms, and approaches. Many also include code examples that you can use with Azure services. The practices apply to any distributed system, whether your host is Azure or a different cloud platform.

Catalog of practices

This table lists various best practices. The **Related pillars or patterns** column contains the following links:

- Cloud development challenges that the practice and related design patterns address.
- Pillars of the Microsoft Azure Well-Architected Framework that the practice focuses on.

PRACTICE	SUMMARY	RELATED PILLARS OR PATTERNS
API design	Design web APIs to support platform independence by using standard protocols and agreed-upon data formats. Promote service evolution so that clients can discover functionality without requiring modification. Improve response times and prevent transient faults by supporting partial responses and providing ways to filter and paginate data.	Design and implementation, Performance efficiency, Operational excellence
API implementation	Implement web APIs to be efficient, responsive, scalable, and available. Make actions idempotent, support content negotiation, and follow the HTTP specification. Handle exceptions, and support the discovery of resources. Provide ways to handle large requests and minimize network traffic.	Design and implementation, Operational excellence
Autoscaling	Design apps to dynamically allocate and de-allocate resources to satisfy performance requirements and minimize costs. Take advantage of Azure Monitor autoscale and the built-in autoscaling that many Azure components offer.	Performance efficiency, Cost optimization
Background jobs	Implement batch jobs, processing tasks, and workflows as background jobs. Use Azure platform services to host these tasks. Trigger tasks with events or schedules, and return results to calling tasks.	Design and implementation, Operational excellence

PRACTICE	SUMMARY	RELATED PILLARS OR PATTERNS
Caching	Improve performance by copying data to fast storage that's close to apps. Cache data that you read often but rarely modify. Manage data expiration and concurrency. See how to populate caches and use the Azure Cache for Redis service.	Data management, Performance efficiency
Content delivery network	Use content delivery networks (CDNs) to efficiently deliver web content to users and reduce load on web apps. Overcome deployment, versioning, security, and resilience challenges.	Data management, Performance efficiency
Data partitioning	Partition data to improve scalability, availability, and performance, and to reduce contention and data storage costs. Use horizontal, vertical, and functional partitioning in efficient ways.	Data management, Performance efficiency, Cost optimization
Data partitioning strategies (by service)	Partition data in Azure SQL Database and Azure Storage services like Azure Table Storage and Azure Blob Storage. Shard your data to distribute loads, reduce latency, and support horizontal scaling.	Data management, Performance efficiency, Cost optimization
Message encoding considerations	Use asynchronous messages to exchange information between system components. Choose the payload structure, encoding format, and serialization library that work best with your data.	Messaging, Security
Monitoring and diagnostics	Track system health, usage, and performance with a monitoring and diagnostics pipeline. Turn monitoring data into alerts, reports, and triggers that help in various situations. Examples include detecting and correcting issues, spotting potential problems, meeting performance guarantees, and fulfilling auditing requirements.	Operational excellence
Retry guidance for specific services	Use, adapt, and extend the retry mechanisms that Azure services and client SDKs offer. Develop a systematic and robust approach for managing temporary issues with connections, operations, and resources.	Design and implementation, Reliability
Transient fault handling	Handle transient faults caused by unavailable networks or resources. Overcome challenges when developing appropriate retry strategies. Avoid duplicating layers of retry code and other anti-patterns.	Design and implementation, Reliability

Next steps

- Web API design
- Web API implementation

Related resources

- Cloud design patterns
- Microsoft Azure Well-Architected Framework

RESTful web API design

11/19/2021 • 28 minutes to read • Edit Online

Most modern web applications expose APIs that clients can use to interact with the application. A well-designed web API should aim to support:

- **Platform independence**. Any client should be able to call the API, regardless of how the API is implemented internally. This requires using standard protocols, and having a mechanism whereby the client and the web service can agree on the format of the data to exchange.
- **Service evolution**. The web API should be able to evolve and add functionality independently from client applications. As the API evolves, existing client applications should continue to function without modification. All functionality should be discoverable so that client applications can fully use it.

This guidance describes issues that you should consider when designing a web API.

What is REST?

In 2000, Roy Fielding proposed Representational State Transfer (REST) as an architectural approach to designing web services. REST is an architectural style for building distributed systems based on hypermedia. REST is independent of any underlying protocol and is not necessarily tied to HTTP. However, most common REST API implementations use HTTP as the application protocol, and this guide focuses on designing REST APIs for HTTP.

A primary advantage of REST over HTTP is that it uses open standards, and does not bind the implementation of the API or the client applications to any specific implementation. For example, a REST web service could be written in ASP.NET, and client applications can use any language or toolset that can generate HTTP requests and parse HTTP responses.

Here are some of the main design principles of RESTful APIs using HTTP:

- REST APIs are designed around *resources*, which are any kind of object, data, or service that can be accessed by the client.
- A resource has an *identifier*, which is a URI that uniquely identifies that resource. For example, the URI for a particular customer order might be:

  ```
  https://adventure-works.com/orders/1
  ```

- Clients interact with a service by exchanging *representations* of resources. Many web APIs use JSON as the exchange format. For example, a GET request to the URI listed above might return this response body:

  ```
  {"orderId":1,"orderValue":99.90,"productId":1,"quantity":1}
  ```

- REST APIs use a uniform interface, which helps to decouple the client and service implementations. For REST APIs built on HTTP, the uniform interface includes using standard HTTP verbs to perform operations on resources. The most common operations are GET, POST, PUT, PATCH, and DELETE.
- REST APIs use a stateless request model. HTTP requests should be independent and may occur in any order, so keeping transient state information between requests is not feasible. The only place where information is stored is in the resources themselves, and each request should be an atomic operation. This constraint enables web services to be highly scalable, because there is no need to retain any affinity

between clients and specific servers. Any server can handle any request from any client. That said, other factors can limit scalability. For example, many web services write to a backend data store, which may be hard to scale out. For more information about strategies to scale out a data store, see Horizontal, vertical, and functional data partitioning.

- REST APIs are driven by hypermedia links that are contained in the representation. For example, the following shows a JSON representation of an order. It contains links to get or update the customer associated with the order.

```
{
    "orderID":3,
    "productID":2,
    "quantity":4,
    "orderValue":16.60,
    "links": [
        {"rel":"product","href":"https://adventure-works.com/customers/3", "action":"GET" },
        {"rel":"product","href":"https://adventure-works.com/customers/3", "action":"PUT" }
    ]
}
```

In 2008, Leonard Richardson proposed the following maturity model for web APIs:

- Level 0: Define one URI, and all operations are POST requests to this URI.
- Level 1: Create separate URIs for individual resources.
- Level 2: Use HTTP methods to define operations on resources.
- Level 3: Use hypermedia (HATEOAS, described below).

Level 3 corresponds to a truly RESTful API according to Fielding's definition. In practice, many published web APIs fall somewhere around level 2.

Organize the API design around resources

Focus on the business entities that the web API exposes. For example, in an e-commerce system, the primary entities might be customers and orders. Creating an order can be achieved by sending an HTTP POST request that contains the order information. The HTTP response indicates whether the order was placed successfully or not. When possible, resource URIs should be based on nouns (the resource) and not verbs (the operations on the resource).

```
https://adventure-works.com/orders // Good

https://adventure-works.com/create-order // Avoid
```

A resource doesn't have to be based on a single physical data item. For example, an order resource might be implemented internally as several tables in a relational database, but presented to the client as a single entity. Avoid creating APIs that simply mirror the internal structure of a database. The purpose of REST is to model entities and the operations that an application can perform on those entities. A client should not be exposed to the internal implementation.

Entities are often grouped together into collections (orders, customers). A collection is a separate resource from the item within the collection, and should have its own URI. For example, the following URI might represent the collection of orders:

```
https://adventure-works.com/orders
```

Sending an HTTP GET request to the collection URI retrieves a list of items in the collection. Each item in the

collection also has its own unique URI. An HTTP GET request to the item's URI returns the details of that item.

Adopt a consistent naming convention in URIs. In general, it helps to use plural nouns for URIs that reference collections. It's a good practice to organize URIs for collections and items into a hierarchy. For example, `/customers` is the path to the customers collection, and `/customers/5` is the path to the customer with ID equal to 5. This approach helps to keep the web API intuitive. Also, many web API frameworks can route requests based on parameterized URI paths, so you could define a route for the path `/customers/{id}`.

Also consider the relationships between different types of resources and how you might expose these associations. For example, the `/customers/5/orders` might represent all of the orders for customer 5. You could also go in the other direction, and represent the association from an order back to a customer with a URI such as `/orders/99/customer`. However, extending this model too far can become cumbersome to implement. A better solution is to provide navigable links to associated resources in the body of the HTTP response message. This mechanism is described in more detail in the section Use HATEOAS to enable navigation to related resources.

In more complex systems, it can be tempting to provide URIs that enable a client to navigate through several levels of relationships, such as `/customers/1/orders/99/products`. However, this level of complexity can be difficult to maintain and is inflexible if the relationships between resources change in the future. Instead, try to keep URIs relatively simple. Once an application has a reference to a resource, it should be possible to use this reference to find items related to that resource. The preceding query can be replaced with the URI `/customers/1/orders` to find all the orders for customer 1, and then `/orders/99/products` to find the products in this order.

> **TIP**
>
> Avoid requiring resource URIs more complex than *collection/item/collection*.

Another factor is that all web requests impose a load on the web server. The more requests, the bigger the load. Therefore, try to avoid "chatty" web APIs that expose a large number of small resources. Such an API may require a client application to send multiple requests to find all of the data that it requires. Instead, you might want to denormalize the data and combine related information into bigger resources that can be retrieved with a single request. However, you need to balance this approach against the overhead of fetching data that the client doesn't need. Retrieving large objects can increase the latency of a request and incur additional bandwidth costs. For more information about these performance antipatterns, see Chatty I/O and Extraneous Fetching.

Avoid introducing dependencies between the web API and the underlying data sources. For example, if your data is stored in a relational database, the web API doesn't need to expose each table as a collection of resources. In fact, that's probably a poor design. Instead, think of the web API as an abstraction of the database. If necessary, introduce a mapping layer between the database and the web API. That way, client applications are isolated from changes to the underlying database scheme.

Finally, it might not be possible to map every operation implemented by a web API to a specific resource. You can handle such *non-resource* scenarios through HTTP requests that invoke a function and return the results as an HTTP response message. For example, a web API that implements simple calculator operations such as add and subtract could provide URIs that expose these operations as pseudo resources and use the query string to specify the parameters required. For example, a GET request to the URI */add?operand1=99&operand2=1* would return a response message with the body containing the value 100. However, only use these forms of URIs sparingly.

Define API operations in terms of HTTP methods

The HTTP protocol defines a number of methods that assign semantic meaning to a request. The common HTTP methods used by most RESTful web APIs are:

- **GET** retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.
- **POST** creates a new resource at the specified URI. The body of the request message provides the details of the new resource. Note that POST can also be used to trigger operations that don't actually create resources.
- **PUT** either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
- **PATCH** performs a partial update of a resource. The request body specifies the set of changes to apply to the resource.
- **DELETE** removes the resource at the specified URI.

The effect of a specific request should depend on whether the resource is a collection or an individual item. The following table summarizes the common conventions adopted by most RESTful implementations using the e-commerce example. Not all of these requests might be implemented—it depends on the specific scenario.

RESOURCE	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

The differences between POST, PUT, and PATCH can be confusing.

- A POST request creates a resource. The server assigns a URI for the new resource, and returns that URI to the client. In the REST model, you frequently apply POST requests to collections. The new resource is added to the collection. A POST request can also be used to submit data for processing to an existing resource, without any new resource being created.

- A PUT request creates a resource *or* updates an existing resource. The client specifies the URI for the resource. The request body contains a complete representation of the resource. If a resource with this URI already exists, it is replaced. Otherwise a new resource is created, if the server supports doing so. PUT requests are most frequently applied to resources that are individual items, such as a specific customer, rather than collections. A server might support updates but not creation via PUT. Whether to support creation via PUT depends on whether the client can meaningfully assign a URI to a resource before it exists. If not, then use POST to create resources and PUT or PATCH to update.

- A PATCH request performs a *partial update* to an existing resource. The client specifies the URI for the resource. The request body specifies a set of *changes* to apply to the resource. This can be more efficient than using PUT, because the client only sends the changes, not the entire representation of the resource. Technically PATCH can also create a new resource (by specifying a set of updates to a "null" resource), if the server supports this.

PUT requests must be idempotent. If a client submits the same PUT request multiple times, the results should always be the same (the same resource will be modified with the same values). POST and PATCH requests are not guaranteed to be idempotent.

Conform to HTTP semantics

This section describes some typical considerations for designing an API that conforms to the HTTP specification. However, it doesn't cover every possible detail or scenario. When in doubt, consult the HTTP specifications.

Media types

As mentioned earlier, clients and servers exchange representations of resources. For example, in a POST request, the request body contains a representation of the resource to create. In a GET request, the response body contains a representation of the fetched resource.

In the HTTP protocol, formats are specified through the use of *media types*, also called MIME types. For non-binary data, most web APIs support JSON (media type = application/json) and possibly XML (media type = application/xml).

The Content-Type header in a request or response specifies the format of the representation. Here is an example of a POST request that includes JSON data:

```
POST https://adventure-works.com/orders HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: 57

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

If the server doesn't support the media type, it should return HTTP status code 415 (Unsupported Media Type).

A client request can include an Accept header that contains a list of media types the client will accept from the server in the response message. For example:

```
GET https://adventure-works.com/orders/2 HTTP/1.1
Accept: application/json
```

If the server cannot match any of the media type(s) listed, it should return HTTP status code 406 (Not Acceptable).

GET methods

A successful GET method typically returns HTTP status code 200 (OK). If the resource cannot be found, the method should return 404 (Not Found).

POST methods

If a POST method creates a new resource, it returns HTTP status code 201 (Created). The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code 200 and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code 204 (No Content) with no response body.

If the client puts invalid data into the request, the server should return HTTP status code 400 (Bad Request). The response body can contain additional information about the error or a link to a URI that provides more details.

PUT methods

If a PUT method creates a new resource, it returns HTTP status code 201 (Created), as with a POST method. If the method updates an existing resource, it returns either 200 (OK) or 204 (No Content). In some cases, it might not be possible to update an existing resource. In that case, consider returning HTTP status code 409 (Conflict).

Consider implementing bulk HTTP PUT operations that can batch updates to multiple resources in a collection. The PUT request should specify the URI of the collection, and the request body should specify the details of the resources to be modified. This approach can help to reduce chattiness and improve performance.

PATCH methods

With a PATCH request, the client sends a set of updates to an existing resource, in the form of a *patch document*.

The server processes the patch document to perform the update. The patch document doesn't describe the whole resource, only a set of changes to apply. The specification for the PATCH method (RFC 5789) doesn't define a particular format for patch documents. The format must be inferred from the media type in the request.

JSON is probably the most common data format for web APIs. There are two main JSON-based patch formats, called *JSON patch* and *JSON merge patch*.

JSON merge patch is somewhat simpler. The patch document has the same structure as the original JSON resource, but includes just the subset of fields that should be changed or added. In addition, a field can be deleted by specifying `null` for the field value in the patch document. (That means merge patch is not suitable if the original resource can have explicit null values.)

For example, suppose the original resource has the following JSON representation:

```
{
    "name":"gizmo",
    "category":"widgets",
    "color":"blue",
    "price":10
}
```

Here is a possible JSON merge patch for this resource:

```
{
    "price":12,
    "color":null,
    "size":"small"
}
```

This tells the server to update `price`, delete `color`, and add `size`, while `name` and `category` are not modified. For the exact details of JSON merge patch, see RFC 7396. The media type for JSON merge patch is `application/merge-patch+json`.

Merge patch is not suitable if the original resource can contain explicit null values, due to the special meaning of `null` in the patch document. Also, the patch document doesn't specify the order that the server should apply the updates. That may or may not matter, depending on the data and the domain. JSON patch, defined in RFC 6902, is more flexible. It specifies the changes as a sequence of operations to apply. Operations include add, remove, replace, copy, and test (to validate values). The media type for JSON patch is `application/json-patch+json`.

Here are some typical error conditions that might be encountered when processing a PATCH request, along with the appropriate HTTP status code.

ERROR CONDITION	HTTP STATUS CODE
The patch document format isn't supported.	415 (Unsupported Media Type)
Malformed patch document.	400 (Bad Request)
The patch document is valid, but the changes can't be applied to the resource in its current state.	409 (Conflict)

DELETE methods

If the delete operation is successful, the web server should respond with HTTP status code 204 (No Content), indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn't exist, the web server can return HTTP 404 (Not Found).

Asynchronous operations

Sometimes a POST, PUT, PATCH, or DELETE operation might require processing that takes a while to complete. If you wait for completion before sending a response to the client, it may cause unacceptable latency. If so, consider making the operation asynchronous. Return HTTP status code 202 (Accepted) to indicate the request was accepted for processing but is not completed.

You should expose an endpoint that returns the status of an asynchronous request, so the client can monitor the status by polling the status endpoint. Include the URI of the status endpoint in the Location header of the 202 response. For example:

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

If the client sends a GET request to this endpoint, the response should contain the current status of the request. Optionally, it could also include an estimated time to completion or a link to cancel the operation.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status":"In progress",
    "link": { "rel":"cancel", "method":"delete", "href":"/api/status/12345" }
}
```

If the asynchronous operation creates a new resource, the status endpoint should return status code 303 (See Other) after the operation completes. In the 303 response, include a Location header that gives the URI of the new resource:

```
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

For more information, see Asynchronous Request-Reply pattern.

Filter and paginate data

Exposing a collection of resources through a single URI can lead to applications fetching large amounts of data when only a subset of the information is required. For example, suppose a client application needs to find all orders with a cost over a specific value. It might retrieve all orders from the */orders* URI and then filter these orders on the client side. Clearly this process is highly inefficient. It wastes network bandwidth and processing power on the server hosting the web API.

Instead, the API can allow passing a filter in the query string of the URI, such as */orders?minCost=n*. The web API is then responsible for parsing and handling the `minCost` parameter in the query string and returning the filtered results on the server side.

GET requests over collection resources can potentially return a large number of items. You should design a web API to limit the amount of data returned by any single request. Consider supporting query strings that specify the maximum number of items to retrieve and a starting offset into the collection. For example:

```
/orders?limit=25&offset=50
```

Also consider imposing an upper limit on the number of items returned, to help prevent Denial of Service attacks. To assist client applications, GET requests that return paginated data should also include some form of metadata that indicate the total number of resources available in the collection.

You can use a similar strategy to sort data as it is fetched, by providing a sort parameter that takes a field name as the value, such as */orders?sort=ProductID*. However, this approach can have a negative effect on caching, because query string parameters form part of the resource identifier used by many cache implementations as the key to cached data.

You can extend this approach to limit the fields returned for each item, if each item contains a large amount of data. For example, you could use a query string parameter that accepts a comma-delimited list of fields, such as */orders?fields=ProductID,Quantity*.

Give all optional parameters in query strings meaningful defaults. For example, set the `limit` parameter to 10 and the `offset` parameter to 0 if you implement pagination, set the sort parameter to the key of the resource if you implement ordering, and set the `fields` parameter to all fields in the resource if you support projections.

Support partial responses for large binary resources

A resource may contain large binary fields, such as files or images. To overcome problems caused by unreliable and intermittent connections and to improve response times, consider enabling such resources to be retrieved in chunks. To do this, the web API should support the Accept-Ranges header for GET requests for large resources. This header indicates that the GET operation supports partial requests. The client application can submit GET requests that return a subset of a resource, specified as a range of bytes.

Also, consider implementing HTTP HEAD requests for these resources. A HEAD request is similar to a GET request, except that it only returns the HTTP headers that describe the resource, with an empty message body. A client application can issue a HEAD request to determine whether to fetch a resource by using partial GET requests. For example:

```
HEAD https://adventure-works.com/products/10?fields=productImage HTTP/1.1
```

Here is an example response message:

```
HTTP/1.1 200 OK

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 4580
```

The Content-Length header gives the total size of the resource, and the Accept-Ranges header indicates that the corresponding GET operation supports partial results. The client application can use this information to retrieve the image in smaller chunks. The first request fetches the first 2500 bytes by using the Range header:

```
GET https://adventure-works.com/products/10?fields=productImage HTTP/1.1
Range: bytes=0-2499
```

The response message indicates that this is a partial response by returning HTTP status code 206. The Content-Length header specifies the actual number of bytes returned in the message body (not the size of the resource), and the Content-Range header indicates which part of the resource this is (bytes 0-2499 out of 4580):

```
HTTP/1.1 206 Partial Content

Accept-Ranges: bytes
Content-Type: image/jpeg
Content-Length: 2500
Content-Range: bytes 0-2499/4580

[...]
```

A subsequent request from the client application can retrieve the remainder of the resource.

Use HATEOAS to enable navigation to related resources

One of the primary motivations behind REST is that it should be possible to navigate the entire set of resources without requiring prior knowledge of the URI scheme. Each HTTP GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks included in the response, and it should also be provided with information that describes the operations available on each of these resources. This principle is known as HATEOAS, or Hypertext as the Engine of Application State. The system is effectively a finite state machine, and the response to each request contains the information necessary to move from one state to another; no other information should be necessary.

> **NOTE**
> Currently there are no general-purpose standards that define how to model the HATEOAS principle. The examples shown in this section illustrate one possible, proprietary solution.

For example, to handle the relationship between an order and a customer, the representation of an order could include links that identify the available operations for the customer of the order. Here is a possible representation:

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"DELETE",
      "types":[]
    }]
}
```

In this example, the `links` array has a set of links. Each link represents an operation on a related entity. The data for each link includes the relationship ("customer"), the URI (`https://adventure-works.com/customers/3`), the HTTP method, and the supported MIME types. This is all the information that a client application needs to be able to invoke the operation.

The `links` array also includes self-referencing information about the resource itself that has been retrieved. These have the relationship *self*.

The set of links that are returned may change, depending on the state of the resource. This is what is meant by hypertext being the "engine of application state."

Versioning a RESTful web API

It is highly unlikely that a web API will remain static. As business requirements change new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. While updating a web API to handle new or differing requirements is a relatively straightforward process, you must consider the effects that such changes will have on client applications consuming the web API. The issue is that although the developer designing and implementing a web API has full

control over that API, the developer does not have the same degree of control over client applications, which may be built by third-party organizations operating remotely. The primary imperative is to enable existing client applications to continue functioning unchanged while allowing new client applications to take advantage of new features and resources.

Versioning enables a web API to indicate the features and resources that it exposes, and a client application can submit requests that are directed to a specific version of a feature or resource. The following sections describe several different approaches, each of which has its own benefits and trade-offs.

No versioning

This is the simplest approach, and may be acceptable for some internal APIs. Significant changes could be represented as new resources or new links. Adding content to existing resources might not present a breaking change as client applications that are not expecting to see this content will ignore it.

For example, a request to the URI `https://adventure-works.com/customers/3` should return the details of a single customer containing `id`, `name`, and `address` fields expected by the client application:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

> **NOTE**
> For simplicity, the example responses shown in this section do not include HATEOAS links.

If the `DateCreated` field is added to the schema of the customer resource, then the response would look like this:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":"1 Microsoft Way Redmond
WA 98053"}
```

Existing client applications might continue functioning correctly if they are capable of ignoring unrecognized fields, while new client applications can be designed to handle this new field. However, if more radical changes to the schema of resources occur (such as removing or renaming fields) or the relationships between resources change then these may constitute breaking changes that prevent existing client applications from functioning correctly. In these situations, you should consider one of the following approaches.

URI versioning

Each time you modify the web API or change the schema of resources, you add a version number to the URI for each resource. The previously existing URIs should continue to operate as before, returning resources that conform to their original schema.

Extending the previous example, if the `address` field is restructured into subfields containing each constituent part of the address (such as `streetAddress`, `city`, `state`, and `zipCode`), this version of the resource could be exposed through a URI containing a version number, such as `https://adventure-works.com/v2/customers/3`:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"1
Microsoft Way","city":"Redmond","state":"WA","zipCode":98053}}
```

This versioning mechanism is very simple but depends on the server routing the request to the appropriate endpoint. However, it can become unwieldy as the web API matures through several iterations and the server has to support a number of different versions. Also, from a purist's point of view, in all cases the client applications are fetching the same data (customer 3), so the URI should not really be different depending on the version. This scheme also complicates implementation of HATEOAS as all links will need to include the version number in their URIs.

Query string versioning

Rather than providing multiple URIs, you can specify the version of the resource by using a parameter within the query string appended to the HTTP request, such as `https://adventure-works.com/customers/3?version=2`. The version parameter should default to a meaningful value such as 1 if it is omitted by older client applications.

This approach has the semantic advantage that the same resource is always retrieved from the same URI, but it depends on the code that handles the request to parse the query string and send back the appropriate HTTP response. This approach also suffers from the same complications for implementing HATEOAS as the URI versioning mechanism.

> **NOTE**
>
> Some older web browsers and web proxies will not cache responses for requests that include a query string in the URI. This can degrade performance for web applications that use a web API and that run from within such a web browser.

Header versioning

Rather than appending the version number as a query string parameter, you could implement a custom header that indicates the version of the resource. This approach requires that the client application adds the appropriate header to any requests, although the code handling the client request could use a default value (version 1) if the version header is omitted. The following examples use a custom header named *Custom-Header*. The value of this header indicates the version of web API.

Version 1:

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=1
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

Version 2:

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Custom-Header: api-version=2
```

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","address":{"streetAddress":"1
Microsoft Way","city":"Redmond","state":"WA","zipCode":98053}}
```

As with the previous two approaches, implementing HATEOAS requires including the appropriate custom header in any links.

Media type versioning

When a client application sends an HTTP GET request to a web server it should stipulate the format of the content that it can handle by using an Accept header, as described earlier in this guidance. Frequently the purpose of the *Accept* header is to allow the client application to specify whether the body of the response should be XML, JSON, or some other common format that the client can parse. However, it is possible to define custom media types that include information enabling the client application to indicate which version of a resource it is expecting.

The following example shows a request that specifies an *Accept* header with the value *application/vnd.adventure-works.v1+json*. The *vnd.adventure-works.v1* element indicates to the web server that it should return version 1 of the resource, while the *json* element specifies that the format of the response body should be JSON:

```
GET https://adventure-works.com/customers/3 HTTP/1.1
Accept: application/vnd.adventure-works.v1+json
```

The code handling the request is responsible for processing the *Accept* header and honoring it as far as possible (the client application may specify multiple formats in the *Accept* header, in which case the web server can choose the most appropriate format for the response body). The web server confirms the format of the data in the response body by using the Content-Type header:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.adventure-works.v1+json; charset=utf-8

{"id":3,"name":"Contoso LLC","address":"1 Microsoft Way Redmond WA 98053"}
```

If the Accept header does not specify any known media types, the web server could generate an HTTP 406 (Not Acceptable) response message or return a message with a default media type.

This approach is arguably the purest of the versioning mechanisms and lends itself naturally to HATEOAS, which can include the MIME type of related data in resource links.

> **NOTE**
>
> When you select a versioning strategy, you should also consider the implications on performance, especially caching on the web server. The URI versioning and Query String versioning schemes are cache-friendly inasmuch as the same URI/query string combination refers to the same data each time.
>
> The Header versioning and Media Type versioning mechanisms typically require additional logic to examine the values in the custom header or the Accept header. In a large-scale environment, many clients using different versions of a web API can result in a significant amount of duplicated data in a server-side cache. This issue can become acute if a client application communicates with a web server through a proxy that implements caching, and that only forwards a request to the web server if it does not currently hold a copy of the requested data in its cache.

Open API Initiative

The Open API Initiative was created by an industry consortium to standardize REST API descriptions across vendors. As part of this initiative, the Swagger 2.0 specification was renamed the OpenAPI Specification (OAS) and brought under the Open API Initiative.

You may want to adopt OpenAPI for your web APIs. Some points to consider:

- The OpenAPI Specification comes with a set of opinionated guidelines on how a REST API should be designed. That has advantages for interoperability, but requires more care when designing your API to conform to the specification.

- OpenAPI promotes a contract-first approach, rather than an implementation-first approach. Contract-first means you design the API contract (the interface) first and then write code that implements the contract.
- Tools like Swagger can generate client libraries or documentation from API contracts. For example, see ASP.NET Web API help pages using Swagger.

More information

- Microsoft REST API guidelines. Detailed recommendations for designing public REST APIs.
- Web API checklist. A useful list of items to consider when designing and implementing a web API.
- Open API Initiative. Documentation and implementation details on Open API.

Web API implementation

11/19/2021 • 46 minutes to read • Edit Online

A carefully designed RESTful web API defines the resources, relationships, and navigation schemes that are accessible to client applications. When you implement and deploy a web API, you should consider the physical requirements of the environment hosting the web API and the way in which the web API is constructed rather than the logical structure of the data. This guidance focuses on best practices for implementing a web API and publishing it to make it available to client applications. For detailed information about web API design, see Web API design.

Processing requests

Consider the following points when you implement the code to handle requests.

GET, PUT, DELETE, HEAD, and PATCH actions should be idempotent

The code that implements these requests should not impose any side-effects. The same request repeated over the same resource should result in the same state. For example, sending multiple DELETE requests to the same URI should have the same effect, although the HTTP status code in the response messages may be different. The first DELETE request might return status code 204 (No Content), while a subsequent DELETE request might return status code 404 (Not Found).

> **NOTE**
>
> The article Idempotency Patterns on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.

POST actions that create new resources should not have unrelated side-effects

If a POST request is intended to create a new resource, the effects of the request should be limited to the new resource (and possibly any directly related resources if there is some sort of linkage involved) For example, in an e-commerce system, a POST request that creates a new order for a customer might also amend inventory levels and generate billing information, but it should not modify information not directly related to the order or have any other side-effects on the overall state of the system.

Avoid implementing chatty POST, PUT, and DELETE operations

Support POST, PUT and DELETE requests over resource collections. A POST request can contain the details for multiple new resources and add them all to the same collection, a PUT request can replace the entire set of resources in a collection, and a DELETE request can remove an entire collection.

The OData support included in ASP.NET Web API 2 provides the ability to batch requests. A client application can package up several web API requests and send them to the server in a single HTTP request, and receive a single HTTP response that contains the replies to each request. For more information, Introducing batch support in Web API and Web API OData.

Follow the HTTP specification when sending a response

A web API must return messages that contain the correct HTTP status code to enable the client to determine how to handle the result, the appropriate HTTP headers so that the client understands the nature of the result, and a suitably formatted body to enable the client to parse the result.

For example, a POST operation should return status code 201 (Created) and the response message should include the URI of the newly created resource in the Location header of the response message.

Support content negotiation

The body of a response message may contain data in a variety of formats. For example, an HTTP GET request could return data in JSON, or XML format. When the client submits a request, it can include an Accept header that specifies the data formats that it can handle. These formats are specified as media types. For example, a client that issues a GET request that retrieves an image can specify an Accept header that lists the media types that the client can handle, such as `image/jpeg, image/gif, image/png`. When the web API returns the result, it should format the data by using one of these media types and specify the format in the Content-Type header of the response.

If the client does not specify an Accept header, then use a sensible default format for the response body. As an example, the ASP.NET Web API framework defaults to JSON for text-based data.

Provide links to support HATEOAS-style navigation and discovery of resources

The HATEOAS approach enables a client to navigate and discover resources from an initial starting point. This is achieved by using links containing URIs; when a client issues an HTTP GET request to obtain a resource, the response should contain URIs that enable a client application to quickly locate any directly related resources. For example, in a web API that supports an e-commerce solution, a customer may have placed many orders. When a client application retrieves the details for a customer, the response should include links that enable the client application to send HTTP GET requests that can retrieve these orders. Additionally, HATEOAS-style links should describe the other operations (POST, PUT, DELETE, and so on) that each linked resource supports together with the corresponding URI to perform each request. This approach is described in more detail in API design.

Currently there are no standards that govern the implementation of HATEOAS, but the following example illustrates one possible approach. In this example, an HTTP GET request that finds the details for a customer returns a response that includes HATEOAS links that reference the orders for that customer:

```
GET https://adventure-works.com/customers/2 HTTP/1.1
Accept: text/json
...
```

```
HTTP/1.1 200 OK
...
Content-Type: application/json; charset=utf-8
...
Content-Length: ...
{"CustomerID":2,"CustomerName":"Bert","Links":[
    {"rel":"self",
    "href":"https://adventure-works.com/customers/2",
    "action":"GET",
    "types":["text/xml","application/json"]},
    {"rel":"self",
    "href":"https://adventure-works.com/customers/2",
    "action":"PUT",
    "types":["application/x-www-form-urlencoded"]},
    {"rel":"self",
    "href":"https://adventure-works.com/customers/2",
    "action":"DELETE",
    "types":[]},
    {"rel":"orders",
    "href":"https://adventure-works.com/customers/2/orders",
    "action":"GET",
    "types":["text/xml","application/json"]},
    {"rel":"orders",
    "href":"https://adventure-works.com/customers/2/orders",
    "action":"POST",
    "types":["application/x-www-form-urlencoded"]}
]}
```

In this example, the customer data is represented by the `Customer` class shown in the following code snippet. The HATEOAS links are held in the `Links` collection property:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string CustomerName { get; set; }
    public List<Link> Links { get; set; }
    ...
}

public class Link
{
    public string Rel { get; set; }
    public string Href { get; set; }
    public string Action { get; set; }
    public string [] Types { get; set; }
}
```

The HTTP GET operation retrieves the customer data from storage and constructs a `Customer` object, and then populates the `Links` collection. The result is formatted as a JSON response message. Each link comprises the following fields:

- The relationship between the object being returned and the object described by the link. In this case `self` indicates that the link is a reference back to the object itself (similar to a `this` pointer in many object-oriented languages), and `orders` is the name of a collection containing the related order information.
- The hyperlink (`Href`) for the object being described by the link in the form of a URI.
- The type of HTTP request (`Action`) that can be sent to this URI.
- The format of any data (`Types`) that should be provided in the HTTP request or that can be returned in the response, depending on the type of the request.

The HATEOAS links shown in the example HTTP response indicate that a client application can perform the following operations:

- An HTTP GET request to the URI `https://adventure-works.com/customers/2` to fetch the details of the customer (again). The data can be returned as XML or JSON.
- An HTTP PUT request to the URI `https://adventure-works.com/customers/2` to modify the details of the customer. The new data must be provided in the request message in x-www-form-urlencoded format.
- An HTTP DELETE request to the URI `https://adventure-works.com/customers/2` to delete the customer. The request does not expect any additional information or return data in the response message body.
- An HTTP GET request to the URI `https://adventure-works.com/customers/2/orders` to find all the orders for the customer. The data can be returned as XML or JSON.
- An HTTP POST request to the URI `https://adventure-works.com/customers/2/orders` to create a new order for this customer. The data must be provided in the request message in x-www-form-urlencoded format.

Handling exceptions

Consider the following points if an operation throws an uncaught exception.

Capture exceptions and return a meaningful response to clients

The code that implements an HTTP operation should provide comprehensive exception handling rather than letting uncaught exceptions propagate to the framework. If an exception makes it impossible to complete the operation successfully, the exception can be passed back in the response message, but it should include a meaningful description of the error that caused the exception. The exception should also include the appropriate HTTP status code rather than simply returning status code 500 for every situation. For example, if a user request

causes a database update that violates a constraint (such as attempting to delete a customer that has outstanding orders), you should return status code 409 (Conflict) and a message body indicating the reason for the conflict. If some other condition renders the request unachievable, you can return status code 400 (Bad Request). You can find a full list of HTTP status codes on the Status code definitions page on the W3C website.

The code example traps different conditions and returns an appropriate response.

```
[HttpDelete]
[Route("customers/{id:int}")]
public IHttpActionResult DeleteCustomer(int id)
{
    try
    {
        // Find the customer to be deleted in the repository
        var customerToDelete = repository.GetCustomer(id);

        // If there is no such customer, return an error response
        // with status code 404 (Not Found)
        if (customerToDelete == null)
        {
            return NotFound();
        }

        // Remove the customer from the repository
        // The DeleteCustomer method returns true if the customer
        // was successfully deleted
        if (repository.DeleteCustomer(id))
        {
            // Return a response message with status code 204 (No Content)
            // To indicate that the operation was successful
            return StatusCode(HttpStatusCode.NoContent);
        }
        else
        {
            // Otherwise return a 400 (Bad Request) error response
            return BadRequest(Strings.CustomerNotDeleted);
        }
    }
    catch
    {
        // If an uncaught exception occurs, return an error response
        // with status code 500 (Internal Server Error)
        return InternalServerError();
    }
}
```

TIP

Do not include information that could be useful to an attacker attempting to penetrate your API.

Many web servers trap error conditions themselves before they reach the web API. For example, if you configure authentication for a web site and the user fails to provide the correct authentication information, the web server should respond with status code 401 (Unauthorized). Once a client has been authenticated, your code can perform its own checks to verify that the client should be able access the requested resource. If this authorization fails, you should return status code 403 (Forbidden).

Handle exceptions consistently and log information about errors

To handle exceptions in a consistent manner, consider implementing a global error handling strategy across the entire web API. You should also incorporate error logging which captures the full details of each exception; this error log can contain detailed information as long as it is not made accessible over the web to clients.

Distinguish between client-side errors and server-side errors

The HTTP protocol distinguishes between errors that occur due to the client application (the HTTP 4xx status codes), and errors that are caused by a mishap on the server (the HTTP 5xx status codes). Make sure that you respect this convention in any error response messages.

Optimizing client-side data access

In a distributed environment such as that involving a web server and client applications, one of the primary sources of concern is the network. This can act as a considerable bottleneck, especially if a client application is frequently sending requests or receiving data. Therefore you should aim to minimize the amount of traffic that flows across the network. Consider the following points when you implement the code to retrieve and maintain data:

Support client-side caching

The HTTP 1.1 protocol supports caching in clients and intermediate servers through which a request is routed by the use of the Cache-Control header. When a client application sends an HTTP GET request to the web API, the response can include a Cache-Control header that indicates whether the data in the body of the response can be safely cached by the client or an intermediate server through which the request has been routed, and for how long before it should expire and be considered out-of-date.

The following example shows an HTTP GET request and the corresponding response that includes a Cache-Control header:

```
GET https://adventure-works.com/orders/2 HTTP/1.1
```

```
HTTP/1.1 200 OK
...
Cache-Control: max-age=600, private
Content-Type: text/json; charset=utf-8
Content-Length: ...
{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

In this example, the Cache-Control header specifies that the data returned should be expired after 600 seconds, and is only suitable for a single client and must not be stored in a shared cache used by other clients (it is *private*). The Cache-Control header could specify *public* rather than *private* in which case the data can be stored in a shared cache, or it could specify *no-store* in which case the data must **not** be cached by the client. The following code example shows how to construct a Cache-Control header in a response message:

```
public class OrdersController : ApiController
{
    ...
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...
        // Create a Cache-Control header for the response
        var cacheControlHeader = new CacheControlHeaderValue();
        cacheControlHeader.Private = true;
        cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
        ...

        // Return a response message containing the order and the cache control header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>(order, this)
        {
            CacheControlHeader = cacheControlHeader
        };
        return response;
    }
    ...
}
```

This code uses a custom `IHttpActionResult` class named `OkResultWithCaching`. This class enables the controller to set the cache header contents:

```
public class OkResultWithCaching<T> : OkNegotiatedContentResult<T>
{
    public OkResultWithCaching(T content, ApiController controller)
        : base(content, controller) { }

    public OkResultWithCaching(T content, IContentNegotiator contentNegotiator, HttpRequestMessage request,
IEnumerable<MediaTypeFormatter> formatters)
        : base(content, contentNegotiator, request, formatters) { }

    public CacheControlHeaderValue CacheControlHeader { get; set; }
    public EntityTagHeaderValue ETag { get; set; }

    public override async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        HttpResponseMessage response;
        try
        {
            response = await base.ExecuteAsync(cancellationToken);
            response.Headers.CacheControl = this.CacheControlHeader;
            response.Headers.ETag = ETag;
        }
        catch (OperationCanceledException)
        {
            response = new HttpResponseMessage(HttpStatusCode.Conflict) {ReasonPhrase = "Operation was
cancelled"};
        }
        return response;
    }
}
```

> **NOTE**
>
> The HTTP protocol also defines the *no-cache* directive for the Cache-Control header. Rather confusingly, this directive does not mean "do not cache" but rather "revalidate the cached information with the server before returning it"; the data can still be cached, but it is checked each time it is used to ensure that it is still current.

Cache management is the responsibility of the client application or intermediate server, but if properly implemented it can save bandwidth and improve performance by removing the need to fetch data that has already been recently retrieved.

The *max-age* value in the Cache-Control header is only a guide and not a guarantee that the corresponding data won't change during the specified time. The web API should set the max-age to a suitable value depending on the expected volatility of the data. When this period expires, the client should discard the object from the cache.

> **NOTE**
>
> Most modern web browsers support client-side caching by adding the appropriate cache-control headers to requests and examining the headers of the results, as described. However, some older browsers will not cache the values returned from a URL that includes a query string. This is not usually an issue for custom client applications which implement their own cache management strategy based on the protocol discussed here.
>
> Some older proxies exhibit the same behavior and might not cache requests based on URLs with query strings. This could be an issue for custom client applications that connect to a web server through such a proxy.

Provide ETags to optimize query processing

When a client application retrieves an object, the response message can also include an *ETag* (Entity Tag). An ETag is an opaque string that indicates the version of a resource; each time a resource changes the ETag is also modified. This ETag should be cached as part of the data by the client application. The following code example shows how to add an ETag as part of the response to an HTTP GET request. This code uses the `GetHashCode` method of an object to generate a numeric value that identifies the object (you can override this method if necessary and generate your own hash using an algorithm such as MD5) :

```
public class OrdersController : ApiController
{
    ...
    public IHttpActionResult FindOrderByID(int id)
    {
        // Find the matching order
        Order order = ...;
        ...

        var hashedOrder = order.GetHashCode();
        string hashedOrderEtag = $"\"{hashedOrder}\"";
        var eTag = new EntityTagHeaderValue(hashedOrderEtag);

        // Return a response message containing the order and the cache control header
        OkResultWithCaching<Order> response = new OkResultWithCaching<Order>(order, this)
        {
            ...,
            ETag = eTag
        };
        return response;
    }
    ...
}
```

The response message posted by the web API looks like this:

```
HTTP/1.1 200 OK
...
Cache-Control: max-age=600, private
Content-Type: text/json; charset=utf-8
ETag: "2147483648"
Content-Length: ...
{"orderID":2,"productID":4,"quantity":2,"orderValue":10.00}
```

TIP

For security reasons, do not allow sensitive data or data returned over an authenticated (HTTPS) connection to be cached.

A client application can issue a subsequent GET request to retrieve the same resource at any time, and if the resource has changed (it has a different ETag) the cached version should be discarded and the new version added to the cache. If a resource is large and requires a significant amount of bandwidth to transmit back to the client, repeated requests to fetch the same data can become inefficient. To combat this, the HTTP protocol defines the following process for optimizing GET requests that you should support in a web API:

- The client constructs a GET request containing the ETag for the currently cached version of the resource referenced in an If-None-Match HTTP header:

  ```
  GET https://adventure-works.com/orders/2 HTTP/1.1
  If-None-Match: "2147483648"
  ```

- The GET operation in the web API obtains the current ETag for the requested data (order 2 in the above example), and compares it to the value in the If-None-Match header.
- If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should return an HTTP response with an empty message body and a status code of 304 (Not Modified).
- If the current ETag for the requested data does not match the ETag provided by the request, then the data has changed and the web API should return an HTTP response with the new data in the message body and a status code of 200 (OK).
- If the requested data no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).
- The client uses the status code to maintain the cache. If the data has not changed (status code 304) then the object can remain cached and the client application should continue to use this version of the object. If the data has changed (status code 200) then the cached object should be discarded and the new one inserted. If the data is no longer available (status code 404) then the object should be removed from the cache.

NOTE

If the response header contains the Cache-Control header no-store then the object should always be removed from the cache regardless of the HTTP status code.

The code below shows the `FindOrderByID` method extended to support the If-None-Match header. Notice that if the If-None-Match header is omitted, the specified order is always retrieved:

```
public class OrdersController : ApiController
{
    [Route("api/orders/{id:int:min(0)}")]
    [HttpGet]
    public IHttpActionResult FindOrderByID(int id)
    {
        try
        {
            // Find the matching order
            Order order = ...;

            // If there is no such order then return NotFound
            if (order == null)
            {
                return NotFound();
            }

            // Generate the ETag for the order
            var hashedOrder = order.GetHashCode();
            string hashedOrderEtag = $"\"{hashedOrder}\"";

            // Create the Cache-Control and ETag headers for the response
            IHttpActionResult response;
            var cacheControlHeader = new CacheControlHeaderValue();
            cacheControlHeader.Public = true;
            cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);
            var eTag = new EntityTagHeaderValue(hashedOrderEtag);

            // Retrieve the If-None-Match header from the request (if it exists)
            var nonMatchEtags = Request.Headers.IfNoneMatch;

            // If there is an ETag in the If-None-Match header and
            // this ETag matches that of the order just retrieved,
            // then create a Not Modified response message
            if (nonMatchEtags.Count > 0 &&
                String.CompareOrdinal(nonMatchEtags.First().Tag, hashedOrderEtag) == 0)
            {
                response = new EmptyResultWithCaching()
                {
                    StatusCode = HttpStatusCode.NotModified,
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag
                };
            }
            // Otherwise create a response message that contains the order details
            else
            {
                response = new OkResultWithCaching<Order>(order, this)
                {
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag
                };
            }

            return response;
        }
        catch
        {
            return InternalServerError();
        }
    }
...
}
```

This example incorporates an additional custom `IHttpActionResult` class named `EmptyResultWithCaching`. This class simply acts as a wrapper around an `HttpResponseMessage` object that does not contain a response body:

```
public class EmptyResultWithCaching : IHttpActionResult
{
    public CacheControlHeaderValue CacheControlHeader { get; set; }
    public EntityTagHeaderValue ETag { get; set; }
    public HttpStatusCode StatusCode { get; set; }
    public Uri Location { get; set; }

    public async Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken)
    {
        HttpResponseMessage response = new HttpResponseMessage(StatusCode);
        response.Headers.CacheControl = this.CacheControlHeader;
        response.Headers.ETag = this.ETag;
        response.Headers.Location = this.Location;
        return response;
    }
}
```

TIP

In this example, the ETag for the data is generated by hashing the data retrieved from the underlying data source. If the ETag can be computed in some other way, then the process can be optimized further and the data only needs to be fetched from the data source if it has changed. This approach is especially useful if the data is large or accessing the data source can result in significant latency (for example, if the data source is a remote database).

Use ETags to Support Optimistic Concurrency

To enable updates over previously cached data, the HTTP protocol supports an optimistic concurrency strategy. If, after fetching and caching a resource, the client application subsequently sends a PUT or DELETE request to change or remove the resource, it should include in If-Match header that references the ETag. The web API can then use this information to determine whether the resource has already been changed by another user since it was retrieved and send an appropriate response back to the client application as follows:

- The client constructs a PUT request containing the new details for the resource and the ETag for the currently cached version of the resource referenced in an If-Match HTTP header. The following example shows a PUT request that updates an order:

  ```
  PUT https://adventure-works.com/orders/1 HTTP/1.1
  If-Match: "2282343857"
  Content-Type: application/x-www-form-urlencoded
  Content-Length: ...
  productID=3&quantity=5&orderValue=250
  ```

- The PUT operation in the web API obtains the current ETag for the requested data (order 1 in the above example), and compares it to the value in the If-Match header.

- If the current ETag for the requested data matches the ETag provided by the request, the resource has not changed and the web API should perform the update, returning a message with HTTP status code 204 (No Content) if it is successful. The response can include Cache-Control and ETag headers for the updated version of the resource. The response should always include the Location header that references the URI of the newly updated resource.

- If the current ETag for the requested data does not match the ETag provided by the request, then the data has been changed by another user since it was fetched and the web API should return an HTTP response with an empty message body and a status code of 412 (Precondition Failed).

- If the resource to be updated no longer exists then the web API should return an HTTP response with the status code of 404 (Not Found).

- The client uses the status code and response headers to maintain the cache. If the data has been updated (status code 204) then the object can remain cached (as long as the Cache-Control header does not specify no-store) but the ETag should be updated. If the data was changed by another user changed (status code 412) or not found (status code 404) then the cached object should be discarded.

The next code example shows an implementation of the PUT operation for the Orders controller:

```
public class OrdersController : ApiController
{
    [HttpPut]
    [Route("api/orders/{id:int}")]
    public IHttpActionResult UpdateExistingOrder(int id, DTOOrder order)
    {
        try
        {
            var baseUri = Constants.GetUriFromConfig();
            var orderToUpdate = this.ordersRepository.GetOrder(id);
            if (orderToUpdate == null)
            {
                return NotFound();
            }

            var hashedOrder = orderToUpdate.GetHashCode();
            string hashedOrderEtag = $"\"{hashedOrder}\"";

            // Retrieve the If-Match header from the request (if it exists)
            var matchEtags = Request.Headers.IfMatch;

            // If there is an ETag in the If-Match header and
            // this ETag matches that of the order just retrieved,
            // or if there is no ETag, then update the Order
            if (((matchEtags.Count > 0 &&
                String.CompareOrdinal(matchEtags.First().Tag, hashedOrderEtag) == 0)) ||
                matchEtags.Count == 0)
            {
                // Modify the order
                orderToUpdate.OrderValue = order.OrderValue;
                orderToUpdate.ProductID = order.ProductID;
                orderToUpdate.Quantity = order.Quantity;

                // Save the order back to the data store
                // ...

                // Create the No Content response with Cache-Control, ETag, and Location headers
                var cacheControlHeader = new CacheControlHeaderValue();
                cacheControlHeader.Private = true;
                cacheControlHeader.MaxAge = new TimeSpan(0, 10, 0);

                hashedOrder = order.GetHashCode();
                hashedOrderEtag = $"\"{hashedOrder}\"";
                var eTag = new EntityTagHeaderValue(hashedOrderEtag);

                var location = new Uri($"{baseUri}/{Constants.ORDERS}/{id}");
                var response = new EmptyResultWithCaching()
                {
                    StatusCode = HttpStatusCode.NoContent,
                    CacheControlHeader = cacheControlHeader,
                    ETag = eTag,
                    Location = location
                };

                return response;
            }

            // Otherwise return a Precondition Failed response
            return StatusCode(HttpStatusCode.PreconditionFailed);
        }
```

```
        }
        catch
        {
            return InternalServerError();
        }
    }
    ...
}
```

> **TIP**
> Use of the If-Match header is entirely optional, and if it is omitted the web API will always attempt to update the specified order, possibly blindly overwriting an update made by another user. To avoid problems due to lost updates, always provide an If-Match header.

Handling large requests and responses

There may be occasions when a client application needs to issue requests that send or receive data that may be several megabytes (or bigger) in size. Waiting while this amount of data is transmitted could cause the client application to become unresponsive. Consider the following points when you need to handle requests that include significant amounts of data:

Optimize requests and responses that involve large objects

Some resources may be large objects or include large fields, such as graphics images or other types of binary data. A web API should support streaming to enable optimized uploading and downloading of these resources.

The HTTP protocol provides the chunked transfer encoding mechanism to stream large data objects back to a client. When the client sends an HTTP GET request for a large object, the web API can send the reply back in piecemeal *chunks* over an HTTP connection. The length of the data in the reply may not be known initially (it might be generated), so the server hosting the web API should send a response message with each chunk that specifies the Transfer-Encoding: Chunked header rather than a Content-Length header. The client application can receive each chunk in turn to build up the complete response. The data transfer completes when the server sends back a final chunk with zero size.

A single request could conceivably result in a massive object that consumes considerable resources. If during the streaming process the web API determines that the amount of data in a request has exceeded some acceptable bounds, it can abort the operation and return a response message with status code 413 (Request Entity Too Large).

You can minimize the size of large objects transmitted over the network by using HTTP compression. This approach helps to reduce the amount of network traffic and the associated network latency, but at the cost of requiring additional processing at the client and the server hosting the web API. For example, a client application that expects to receive compressed data can include an Accept-Encoding: gzip request header (other data compression algorithms can also be specified). If the server supports compression it should respond with the content held in gzip format in the message body and the Content-Encoding: gzip response header.

You can combine encoded compression with streaming; compress the data first before streaming it, and specify the gzip content encoding and chunked transfer encoding in the message headers. Also note that some web servers (such as Internet Information Server) can be configured to automatically compress HTTP responses regardless of whether the web API compresses the data or not.

Implement partial responses for clients that do not support asynchronous operations

As an alternative to asynchronous streaming, a client application can explicitly request data for large objects in chunks, known as partial responses. The client application sends an HTTP HEAD request to obtain information about the object. If the web API supports partial responses it should respond to the HEAD request with a response message that contains an Accept-Ranges header and a Content-Length header that indicates the total

size of the object, but the body of the message should be empty. The client application can use this information to construct a series of GET requests that specify a range of bytes to receive. The web API should return a response message with HTTP status 206 (Partial Content), a Content-Length header that specifies the actual amount of data included in the body of the response message, and a Content-Range header that indicates which part (such as bytes 4000 to 8000) of the object this data represents.

HTTP HEAD requests and partial responses are described in more detail in API design.

Avoid sending unnecessary 100-Continue status messages in client applications

A client application that is about to send a large amount of data to a server may determine first whether the server is actually willing to accept the request. Prior to sending the data, the client application can submit an HTTP request with an Expect: 100-Continue header, a Content-Length header that indicates the size of the data, but an empty message body. If the server is willing to handle the request, it should respond with a message that specifies the HTTP status 100 (Continue). The client application can then proceed and send the complete request including the data in the message body.

If you are hosting a service by using IIS, the HTTP.sys driver automatically detects and handles Expect: 100-Continue headers before passing requests to your web application. This means that you are unlikely to see these headers in your application code, and you can assume that IIS has already filtered any messages that it deems to be unfit or too large.

If you are building client applications by using the .NET Framework, then all POST and PUT messages will first send messages with Expect: 100-Continue headers by default. As with the server-side, the process is handled transparently by the .NET Framework. However, this process results in each POST and PUT request causing two round-trips to the server, even for small requests. If your application is not sending requests with large amounts of data, you can disable this feature by using the `ServicePointManager` class to create `ServicePoint` objects in the client application. A `ServicePoint` object handles the connections that the client makes to a server based on the scheme and host fragments of URIs that identify resources on the server. You can then set the `Expect100Continue` property of the `ServicePoint` object to false. All subsequent POST and PUT requests made by the client through a URI that matches the scheme and host fragments of the `ServicePoint` object will be sent without Expect: 100-Continue headers. The following code shows how to configure a `ServicePoint` object that configures all requests sent to URIs with a scheme of `http` and a host of `www.contoso.com`.

```
Uri uri = new Uri("https://www.contoso.com/");
ServicePoint sp = ServicePointManager.FindServicePoint(uri);
sp.Expect100Continue = false;
```

You can also set the static `Expect100Continue` property of the `ServicePointManager` class to specify the default value of this property for all subsequently created ServicePoint objects.

Support pagination for requests that may return large numbers of objects

If a collection contains a large number of resources, issuing a GET request to the corresponding URI could result in significant processing on the server hosting the web API affecting performance, and generate a significant amount of network traffic resulting in increased latency.

To handle these cases, the web API should support query strings that enable the client application to refine requests or fetch data in more manageable, discrete blocks (or pages). The code below shows the `GetAllOrders` method in the `Orders` controller. This method retrieves the details of orders. If this method was unconstrained, it could conceivably return a large amount of data. The `limit` and `offset` parameters are intended to reduce the volume of data to a smaller subset, in this case only the first 10 orders by default:

```
public class OrdersController : ApiController
{
    ...
    [Route("api/orders")]
    [HttpGet]
    public IEnumerable<Order> GetAllOrders(int limit=10, int offset=0)
    {
        // Find the number of orders specified by the limit parameter
        // starting with the order specified by the offset parameter
        var orders = ...
        return orders;
    }
    ...
}
```

A client application can issue a request to retrieve 30 orders starting at offset 50 by using the URI `https://www.adventure-works.com/api/orders?limit=30&offset=50`.

> **TIP**
> Avoid enabling client applications to specify query strings that result in a URI that is more than 2000 characters long. Many web clients and servers cannot handle URIs that are this long.

Maintaining responsiveness, scalability, and availability

The same web API might be used by many client applications running anywhere in the world. It is important to ensure that the web API is implemented to maintain responsiveness under a heavy load, to be scalable to support a highly varying workload, and to guarantee availability for clients that perform business-critical operations. Consider the following points when determining how to meet these requirements:

Provide asynchronous support for long-running requests

A request that might take a long time to process should be performed without blocking the client that submitted the request. The web API can perform some initial checking to validate the request, initiate a separate task to perform the work, and then return a response message with HTTP code 202 (Accepted). The task could run asynchronously as part of the web API processing, or it could be offloaded to a background task.

The web API should also provide a mechanism to return the results of the processing to the client application. You can achieve this by providing a polling mechanism for client applications to periodically query whether the processing has finished and obtain the result, or enabling the web API to send a notification when the operation has completed.

You can implement a simple polling mechanism by providing a *polling* URI that acts as a virtual resource using the following approach:

1. The client application sends the initial request to the web API.
2. The web API stores information about the request in a table held in table storage or Microsoft Azure Cache, and generates a unique key for this entry, possibly in the form of a GUID.
3. The web API initiates the processing as a separate task. The web API records the state of the task in the table as *Running*.
4. The web API returns a response message with HTTP status code 202 (Accepted), and the GUID of the table entry in the body of the message.
5. When the task has completed, the web API stores the results in the table, and sets the state of the task to *Complete*. Note that if the task fails, the web API could also store information about the failure and set the status to *Failed*.
6. While the task is running, the client can continue performing its own processing. It can periodically send a

request to the URI */polling/{guid}* where *{guid}* is the GUID returned in the 202 response message by the web API.

7. The web API at the */polling/{guid}* URI queries the state of the corresponding task in the table and returns a response message with HTTP status code 200 (OK) containing this state (*Running*, *Complete*, or *Failed*). If the task has completed or failed, the response message can also include the results of the processing or any information available about the reason for the failure.

Options for implementing notifications include:

- Using a notification hub to push asynchronous responses to client applications. For more information, see Send notifications to specific users by using Azure Notification Hubs.
- Using the Comet model to retain a persistent network connection between the client and the server hosting the web API, and using this connection to push messages from the server back to the client. The MSDN magazine article Building a Simple Comet Application in the Microsoft .NET Framework describes an example solution.
- Using SignalR to push data in real time from the web server to the client over a persistent network connection. SignalR is available for ASP.NET web applications as a NuGet package. You can find more information on the ASP.NET SignalR website.

Ensure that each request is stateless

Each request should be considered atomic. There should be no dependencies between one request made by a client application and any subsequent requests submitted by the same client. This approach assists in scalability; instances of the web service can be deployed on a number of servers. Client requests can be directed at any of these instances and the results should always be the same. It also improves availability for a similar reason; if a web server fails requests can be routed to another instance (by using Azure Traffic Manager) while the server is restarted with no ill effects on client applications.

Track clients and implement throttling to reduce the chances of DOS attacks

If a specific client makes a large number of requests within a given period of time it might monopolize the service and affect the performance of other clients. To mitigate this issue, a web API can monitor calls from client applications either by tracking the IP address of all incoming requests or by logging each authenticated access. You can use this information to limit resource access. If a client exceeds a defined limit, the web API can return a response message with status 503 (Service Unavailable) and include a Retry-After header that specifies when the client can send the next request without it being declined. This strategy can help to reduce the chances of a Denial Of Service (DOS) attack from a set of clients stalling the system.

Manage persistent HTTP connections carefully

The HTTP protocol supports persistent HTTP connections where they are available. The HTTP 1.0 specification added the Connection:Keep-Alive header that enables a client application to indicate to the server that it can use the same connection to send subsequent requests rather than opening new ones. The connection closes automatically if the client does not reuse the connection within a period defined by the host. This behavior is the default in HTTP 1.1 as used by Azure services, so there is no need to include Keep-Alive headers in messages.

Keeping a connection open can help to improve responsiveness by reducing latency and network congestion, but it can be detrimental to scalability by keeping unnecessary connections open for longer than required, limiting the ability of other concurrent clients to connect. It can also affect battery life if the client application is running on a mobile device; if the application only makes occasional requests to the server, maintaining an open connection can cause the battery to drain more quickly. To ensure that a connection is not made persistent with HTTP 1.1, the client can include a Connection:Close header with messages to override the default behavior. Similarly, if a server is handling a very large number of clients it can include a Connection:Close header in response messages which should close the connection and save server resources.

> **NOTE**
> Persistent HTTP connections are a purely optional feature to reduce the network overhead associated with repeatedly establishing a communications channel. Neither the web API nor the client application should depend on a persistent HTTP connection being available. Do not use persistent HTTP connections to implement Comet-style notification systems; instead you should use sockets (or web sockets if available) at the TCP layer. Finally, note Keep-Alive headers are of limited use if a client application communicates with a server via a proxy; only the connection with the client and the proxy will be persistent.

Publishing and managing a web API

To make a web API available for client applications, the web API must be deployed to a host environment. This environment is typically a web server, although it may be some other type of host process. You should consider the following points when publishing a web API:

- All requests must be authenticated and authorized, and the appropriate level of access control must be enforced.
- A commercial web API might be subject to various quality guarantees concerning response times. It is important to ensure that host environment is scalable if the load can vary significantly over time.
- It may be necessary to meter requests for monetization purposes.
- It might be necessary to regulate the flow of traffic to the web API, and implement throttling for specific clients that have exhausted their quotas.
- Regulatory requirements might mandate logging and auditing of all requests and responses.
- To ensure availability, it may be necessary to monitor the health of the server hosting the web API and restart it if necessary.

It is useful to be able to decouple these issues from the technical issues concerning the implementation of the web API. For this reason, consider creating a façade, running as a separate process and that routes requests to the web API. The façade can provide the management operations and forward validated requests to the web API. Using a façade can also bring many functional advantages, including:

- Acting as an integration point for multiple web APIs.
- Transforming messages and translating communications protocols for clients built by using varying technologies.
- Caching requests and responses to reduce load on the server hosting the web API.

Testing a web API

A web API should be tested as thoroughly as any other piece of software. You should consider creating unit tests to validate the functionality.

The nature of a web API brings its own additional requirements to verify that it operates correctly. You should pay particular attention to the following aspects:

- Test all routes to verify that they invoke the correct operations. Be especially aware of HTTP status code 405 (Method Not Allowed) being returned unexpectedly as this can indicate a mismatch between a route and the HTTP methods (GET, POST, PUT, DELETE) that can be dispatched to that route.

  Send HTTP requests to routes that do not support them, such as submitting a POST request to a specific resource (POST requests should only be sent to resource collections). In these cases, the only valid response *should* be status code 405 (Not Allowed).

- Verify that all routes are protected properly and are subject to the appropriate authentication and authorization checks.

> **NOTE**
> Some aspects of security such as user authentication are most likely to be the responsibility of the host environment rather than the web API, but it is still necessary to include security tests as part of the deployment process.

- Test the exception handling performed by each operation and verify that an appropriate and meaningful HTTP response is passed back to the client application.
- Verify that request and response messages are well-formed. For example, if an HTTP POST request contains the data for a new resource in x-www-form-urlencoded format, confirm that the corresponding operation correctly parses the data, creates the resources, and returns a response containing the details of the new resource, including the correct Location header.
- Verify all links and URIs in response messages. For example, an HTTP POST message should return the URI of the newly created resource. All HATEOAS links should be valid.
- Ensure that each operation returns the correct status codes for different combinations of input. For example:
  - If a query is successful, it should return status code 200 (OK)
  - If a resource is not found, the operation should return HTTP status code 404 (Not Found).
  - If the client sends a request that successfully deletes a resource, the status code should be 204 (No Content).
  - If the client sends a request that creates a new resource, the status code should be 201 (Created).

Watch out for unexpected response status codes in the 5xx range. These messages are usually reported by the host server to indicate that it was unable to fulfill a valid request.

- Test the different request header combinations that a client application can specify and ensure that the web API returns the expected information in response messages.
- Test query strings. If an operation can take optional parameters (such as pagination requests), test the different combinations and order of parameters.
- Verify that asynchronous operations complete successfully. If the web API supports streaming for requests that return large binary objects (such as video or audio), ensure that client requests are not blocked while the data is streamed. If the web API implements polling for long-running data modification operations, verify that the operations report their status correctly as they proceed.

You should also create and run performance tests to check that the web API operates satisfactorily under duress. You can build a web performance and load test project by using Visual Studio Ultimate. For more information, see Run performance tests on an application before a release.

Using Azure API Management

On Azure, consider using Azure API Management to publish and manage a web API. Using this facility, you can generate a service that acts as a façade for one or more web APIs. The service is itself a scalable web service that you can create and configure by using the Azure portal. You can use this service to publish and manage a web API as follows:

1. Deploy the web API to a website, Azure cloud service, or Azure virtual machine.
2. Connect the API management service to the web API. Requests sent to the URL of the management API are mapped to URIs in the web API. The same API management service can route requests to more than one web API. This enables you to aggregate multiple web APIs into a single management service.

Similarly, the same web API can be referenced from more than one API management service if you need to restrict or partition the functionality available to different applications.

> **NOTE**
> The URIs in HATEOAS links generated as part of the response for HTTP GET requests should reference the URL of the API management service and not the web server hosting the web API.

3. For each web API, specify the HTTP operations that the web API exposes together with any optional parameters that an operation can take as input. You can also configure whether the API management service should cache the response received from the web API to optimize repeated requests for the same data. Record the details of the HTTP responses that each operation can generate. This information is used to generate documentation for developers, so it is important that it is accurate and complete.

   You can either define operations manually using the wizards provided by the Azure portal, or you can import them from a file containing the definitions in WADL or Swagger format.

4. Configure the security settings for communications between the API management service and the web server hosting the web API. The API management service currently supports Basic authentication and mutual authentication using certificates, and OAuth 2.0 user authorization.

5. Create a product. A product is the unit of publication; you add the web APIs that you previously connected to the management service to the product. When the product is published, the web APIs become available to developers.

   > **NOTE**
   > Prior to publishing a product, you can also define user-groups that can access the product and add users to these groups. This gives you control over the developers and applications that can use the web API. If a web API is subject to approval, prior to being able to access it a developer must send a request to the product administrator. The administrator can grant or deny access to the developer. Existing developers can also be blocked if circumstances change.

6. Configure policies for each web API. Policies govern aspects such as whether cross-domain calls should be allowed, how to authenticate clients, whether to convert between XML and JSON data formats transparently, whether to restrict calls from a given IP range, usage quotas, and whether to limit the call rate. Policies can be applied globally across the entire product, for a single web API in a product, or for individual operations in a web API.

For more information, see the API Management documentation.

> **TIP**
> Azure provides the Azure Traffic Manager which enables you to implement failover and load-balancing, and reduce latency across multiple instances of a web site hosted in different geographic locations. You can use Azure Traffic Manager in conjunction with the API Management Service; the API Management Service can route requests to instances of a web site through Azure Traffic Manager. For more information, see Traffic Manager routing methods.
>
> In this structure, if you are using custom DNS names for your web sites, you should configure the appropriate CNAME record for each web site to point to the DNS name of the Azure Traffic Manager web site.

Supporting client-side developers

Developers constructing client applications typically require information on how to access the web API, and

documentation concerning the parameters, data types, return types, and return codes that describe the different requests and responses between the web service and the client application.

Document the REST operations for a web API

The Azure API Management Service includes a developer portal that describes the REST operations exposed by a web API. When a product has been published it appears on this portal. Developers can use this portal to sign up for access; the administrator can then approve or deny the request. If the developer is approved, they are assigned a subscription key that is used to authenticate calls from the client applications that they develop. This key must be provided with each web API call otherwise it will be rejected.

This portal also provides:

- Documentation for the product, listing the operations that it exposes, the parameters required, and the different responses that can be returned. Note that this information is generated from the details provided in step 3 in the list in the Publishing a web API by using the Microsoft Azure API Management Service section.
- Code snippets that show how to invoke operations from several languages, including JavaScript, C#, Java, Ruby, Python, and PHP.
- A developers' console that enables a developer to send an HTTP request to test each operation in the product and view the results.
- A page where the developer can report any issues or problems found.

The Azure portal enables you to customize the developer portal to change the styling and layout to match the branding of your organization.

Implement a client SDK

Building a client application that invokes REST requests to access a web API requires writing a significant amount of code to construct each request and format it appropriately, send the request to the server hosting the web service, and parse the response to work out whether the request succeeded or failed and extract any data returned. To insulate the client application from these concerns, you can provide an SDK that wraps the REST interface and abstracts these low-level details inside a more functional set of methods. A client application uses these methods, which transparently convert calls into REST requests and then convert the responses back into method return values. This is a common technique that is implemented by many services, including the Azure SDK.

Creating a client-side SDK is a considerable undertaking as it has to be implemented consistently and tested carefully. However, much of this process can be made mechanical, and many vendors supply tools that can automate many of these tasks.

Monitoring a web API

Depending on how you have published and deployed your web API you can monitor the web API directly, or you can gather usage and health information by analyzing the traffic that passes through the API Management service.

Monitoring a web API directly

If you have implemented your web API by using the ASP.NET Web API template (either as a Web API project or as a Web role in an Azure cloud service) and Visual Studio 2013, you can gather availability, performance, and usage data by using ASP.NET Application Insights. Application Insights is a package that transparently tracks and records information about requests and responses when the web API is deployed to the cloud; once the package is installed and configured, you don't need to amend any code in your web API to use it. When you deploy the web API to an Azure web site, all traffic is examined and the following statistics are gathered:

- Server response time.
- Number of server requests and the details of each request.

- The top slowest requests in terms of average response time.
- The details of any failed requests.
- The number of sessions initiated by different browsers and user agents.
- The most frequently viewed pages (primarily useful for web applications rather than web APIs).
- The different user roles accessing the web API.

You can view this data in real time in the Azure portal. You can also create web tests that monitor the health of the web API. A web test sends a periodic request to a specified URI in the web API and captures the response. You can specify the definition of a successful response (such as HTTP status code 200), and if the request does not return this response you can arrange for an alert to be sent to an administrator. If necessary, the administrator can restart the server hosting the web API if it has failed.

For more information, see Application Insights - Get started with ASP.NET.

Monitoring a web API through the API Management Service

If you have published your web API by using the API Management service, the API Management page on the Azure portal contains a dashboard that enables you to view the overall performance of the service. The Analytics page enables you to drill down into the details of how the product is being used. This page contains the following tabs:

- **Usage**. This tab provides information about the number of API calls made and the bandwidth used to handle these calls over time. You can filter usage details by product, API, and operation.
- **Health**. This tab enables you to view the outcome of API requests (the HTTP status codes returned), the effectiveness of the caching policy, the API response time, and the service response time. Again, you can filter health data by product, API, and operation.
- **Activity**. This tab provides a text summary of the numbers of successful calls, failed calls, blocked calls, average response time, and response times for each product, web API, and operation. This page also lists the number of calls made by each developer.
- **At a glance**. This tab displays a summary of the performance data, including the developers responsible for making the most API calls, and the products, web APIs, and operations that received these calls.

You can use this information to determine whether a particular web API or operation is causing a bottleneck, and if necessary scale the host environment and add more servers. You can also ascertain whether one or more applications are using a disproportionate volume of resources and apply the appropriate policies to set quotas and limit call rates.

> **NOTE**
>
> You can change the details for a published product, and the changes are applied immediately. For example, you can add or remove an operation from a web API without requiring that you republish the product that contains the web API.

More information

- ASP.NET Web API OData contains examples and further information on implementing an OData web API by using ASP.NET.
- Introducing batch support in Web API and Web API OData describes how to implement batch operations in a web API by using OData.
- Idempotency patterns on Jonathan Oliver's blog provides an overview of idempotency and how it relates to data management operations.
- Status code definitions on the W3C website contains a full list of HTTP status codes and their descriptions.
- Run background tasks with WebJobs provides information and examples on using WebJobs to perform background operations.

- Azure Notification Hubs notify users shows how to use an Azure Notification Hub to push asynchronous responses to client applications.
- API Management describes how to publish a product that provides controlled and secure access to a web API.
- Azure API Management REST API reference describes how to use the API Management REST API to build custom management applications.
- Traffic Manager routing methods summarizes how Azure Traffic Manager can be used to load-balance requests across multiple instances of a website hosting a web API.
- Application Insights - Get started with ASP.NET provides detailed information on installing and configuring Application Insights in an ASP.NET Web API project.

Autoscaling

11/19/2021 • 15 minutes to read • Edit Online

Autoscaling is the process of dynamically allocating resources to match performance requirements. As the volume of work grows, an application may need additional resources to maintain the desired performance levels and satisfy service-level agreements (SLAs). As demand slackens and the additional resources are no longer needed, they can be de-allocated to minimize costs.

Autoscaling takes advantage of the elasticity of cloud-hosted environments while easing management overhead. It reduces the need for an operator to continually monitor the performance of a system and make decisions about adding or removing resources.

There are two main ways that an application can scale:

- **Vertical scaling**, also called scaling up and down, means changing the capacity of a resource. For example, you could move an application to a larger VM size. Vertical scaling often requires making the system temporarily unavailable while it is being redeployed. Therefore, it's less common to automate vertical scaling.
- **Horizontal scaling**, also called scaling out and in, means adding or removing instances of a resource. The application continues running without interruption as new resources are provisioned. When the provisioning process is complete, the solution is deployed on these additional resources. If demand drops, the additional resources can be shut down cleanly and deallocated.

Many cloud-based systems, including Microsoft Azure, support automatic horizontal scaling. The rest of this article focuses on horizontal scaling.

> **NOTE**
>
> Autoscaling mostly applies to compute resources. While it's possible to horizontally scale a database or message queue, this usually involves data partitioning, which is generally not automated.

Autoscaling components

An autoscaling strategy typically involves the following pieces:

- Instrumentation and monitoring systems at the application, service, and infrastructure levels. These systems capture key metrics, such as response times, queue lengths, CPU utilization, and memory usage.
- Decision-making logic that evaluates these metrics against predefined thresholds or schedules, and decides whether to scale.
- Components that scale the system.
- Testing, monitoring, and tuning of the autoscaling strategy to ensure that it functions as expected.

Azure provides built-in autoscaling mechanisms that address common scenarios. If a particular service or technology does not have built-in autoscaling functionality, or if you have specific autoscaling requirements beyond its capabilities, you might consider a custom implementation. A custom implementation would collect operational and system metrics, analyze the metrics, and then scale resources accordingly.

Configure autoscaling for an Azure solution

Azure provides built-in autoscaling for most compute options.

- **Azure Virtual Machines** autoscale via virtual machine scale sets, which manage a set of Azure virtual machines as a group. See How to use automatic scaling and virtual machine scale sets.
- **Service Fabric** also supports autoscaling through virtual machine scale sets. Every node type in a Service Fabric cluster is set up as a separate virtual machine scale set. That way, each node type can be scaled in or out independently. See Scale a Service Fabric cluster in or out using autoscale rules.
- **Azure App Service** has built-in autoscaling. Autoscale settings apply to all of the apps within an App Service. See Scale instance count manually or automatically.
- **Azure Cloud Services** has built-in autoscaling at the role level. See How to configure auto scaling for a Cloud Service in the portal.

These compute options all use Azure Monitor autoscale to provide a common set of autoscaling functionality.

- **Azure Functions** differs from the previous compute options, because you don't need to configure any autoscale rules. Instead, Azure Functions automatically allocates compute power when your code is running, scaling out as necessary to handle load. For more information, see Choose the correct hosting plan for Azure Functions.

Finally, a custom autoscaling solution can sometimes be useful. For example, you could use Azure diagnostics and application-based metrics, along with custom code to monitor and export the application metrics. Then you could define custom rules based on these metrics, and use Resource Manager REST APIs to trigger autoscaling. However, a custom solution is not simple to implement, and should be considered only if none of the previous approaches can fulfill your requirements.

Use the built-in autoscaling features of the platform, if they meet your requirements. If not, carefully consider whether you really need more complex scaling features. Examples of additional requirements may include more granularity of control, different ways to detect trigger events for scaling, scaling across subscriptions, and scaling other types of resources.

Use Azure Monitor autoscale

Azure Monitor autoscale provide a common set of autoscaling functionality for virtual machine scale sets, Azure App Service, and Azure Cloud Service. Scaling can be performed on a schedule, or based on a runtime metric, such as CPU or memory usage.

Examples:

- Scale out to 10 instances on weekdays, and scale in to 4 instances on Saturday and Sunday.
- Scale out by one instance if average CPU usage is above 70%, and scale in by one instance if CPU usage falls below 50%.
- Scale out by one instance if the number of messages in a queue exceeds a certain threshold.

Scale up the resource when load increases to ensure availability. Similarly, at times of low usage, scale down, so you can optimize cost. Always use a scale-out and scale-in rule combination. Otherwise, the autoscaling takes place only in one direction until it reaches the threshold (maximum or minimum instance counts) set in the profile.

Select a default instance count that's safe for your workload. It's scaled based on that value if maximum or minimum instance counts are not set.

For a list of built-in metrics, see Azure Monitor autoscaling common metrics. You can also implement custom metrics by using Application Insights.

You can configure autoscaling by using PowerShell, the Azure CLI, an Azure Resource Manager template, or the Azure portal. For more detailed control, use the Azure Resource Manager REST API. The Azure Monitoring

Service Management Library and the Microsoft Insights Library (in preview) are SDKs that allow collecting metrics from different resources, and perform autoscaling by making use of the REST APIs. For resources where Azure Resource Manager support isn't available, or if you are using Azure Cloud Services, the Service Management REST API can be used for autoscaling. In all other cases, use Azure Resource Manager.

Consider the following points when using Azure autoscale:

- Consider whether you can predict the load on the application accurately enough to use scheduled autoscaling, adding and removing instances to meet anticipated peaks in demand. If this isn't possible, use reactive autoscaling based on runtime metrics, in order to handle unpredictable changes in demand. Typically, you can combine these approaches. For example, create a strategy that adds resources based on a schedule of the times when you know the application is busiest. This helps to ensure that capacity is available when required, without any delay from starting new instances. For each scheduled rule, define metrics that allow reactive autoscaling during that period to ensure that the application can handle sustained but unpredictable peaks in demand.

- It's often difficult to understand the relationship between metrics and capacity requirements, especially when an application is initially deployed. Provision a little extra capacity at the beginning, and then monitor and tune the autoscaling rules to bring the capacity closer to the actual load.

- Configure the autoscaling rules, and then monitor the performance of your application over time. Use the results of this monitoring to adjust the way in which the system scales if necessary. However, keep in mind that autoscaling is not an instantaneous process. It takes time to react to a metric such as average CPU utilization exceeding (or falling below) a specified threshold.

- Autoscaling rules that use a detection mechanism based on a measured trigger attribute (such as CPU usage or queue length) use an aggregated value over time, rather than instantaneous values, to trigger an autoscaling action. By default, the aggregate is an average of the values. This prevents the system from reacting too quickly, or causing rapid oscillation. It also allows time for new instances that are automatically started to settle into running mode, preventing additional autoscaling actions from occurring while the new instances are starting up. For Azure Cloud Services and Azure Virtual Machines, the default period for the aggregation is 45 minutes, so it can take up to this period of time for the metric to trigger autoscaling in response to spikes in demand. You can change the aggregation period by using the SDK, but periods of less than 25 minutes may cause unpredictable results. For Web Apps, the averaging period is much shorter, allowing new instances to be available in about five minutes after a change to the average trigger measure.

- Avoid *flapping* where scale-in and scale-out actions continually go back and forth. Suppose there are two instances, and upper limit is 80% CPU, lower limit is 60%. When the load is at 85%, another instance is added. After some time, the load decreases to 60%. Before scaling in, the autoscale service calculates the distribution of total load (of three instances) when an instance is removed, taking it to 90%. This means it would have to scale out again immediately. So, it skips scaling-in and you might never see the expected scaling results.

  The flapping situation can be controlled by choosing an adequate margin between the scale-out and scale-in thresholds.

- Manual scaling is reset by maximum and minimum number of instances used for autoscaling. If you manually update the instance count to a value higher or lower than the maximum value, the autoscale engine automatically scales back to the minimum (if lower) or the maximum (if higher). For example, you set the range between 3 and 6. If you have one running instance, the autoscale engine scales to three instances on its next run. Likewise, if you manually set the scale to eight instances, on the next run autoscale will scale it back to six instances on its next run. Manual scaling is temporary unless you reset the autoscale rules as well.

- The autoscale engine processes only one profile at a time. If a condition is not met, then it checks for the

next profile. Keep key metrics out of the default profile because that profile is checked last. Within a profile, you can have multiple rules. On scale-out, autoscale runs if any rule is met. On scale-in, autoscale require all rules to be met.

For details about how Azure Monitor scales, see Best practices for Autoscale.

- If you configure autoscaling using the SDK rather than the portal, you can specify a more detailed schedule during which the rules are active. You can also create your own metrics and use them with or without any of the existing ones in your autoscaling rules. For example, you may wish to use alternative counters, such as the number of requests per second or the average memory availability, or use custom counters to measure specific business processes.
- When autoscaling Service Fabric, the node types in your cluster are made of virtual machine scale sets at the back end, so you need to set up autoscale rules for each node type. Take into account the number of nodes that you must have before you set up autoscaling. The minimum number of nodes that you must have for the primary node type is driven by the reliability level you have chosen. For more information, see scale a Service Fabric cluster in or out using autoscale rules.
- You can use the portal to link resources such as SQL Database instances and queues to a Cloud Service instance. This allows you to more easily access the separate manual and automatic scaling configuration options for each of the linked resources. For more information, see How to: Link a resource to a cloud service.
- When you configure multiple policies and rules, they could conflict with each other. Autoscale uses the following conflict resolution rules to ensure that there is always a sufficient number of instances running:
  - Scale-out operations always take precedence over scale-in operations.
  - When scale-out operations conflict, the rule that initiates the largest increase in the number of instances takes precedence.
  - When scale in operations conflict, the rule that initiates the smallest decrease in the number of instances takes precedence.
- In an App Service Environment, any worker pool or front-end metrics can be used to define autoscale rules. For more information, see Autoscaling and App Service Environment.

Application design considerations

Autoscaling isn't an instant solution. Simply adding resources to a system or running more instances of a process doesn't guarantee that the performance of the system will improve. Consider the following points when designing an autoscaling strategy:

- The system must be designed to be horizontally scalable. Avoid making assumptions about instance affinity; do not design solutions that require that the code is always running in a specific instance of a process. When scaling a cloud service or web site horizontally, don't assume that a series of requests from the same source will always be routed to the same instance. For the same reason, design services to be stateless to avoid requiring a series of requests from an application to always be routed to the same instance of a service. When designing a service that reads messages from a queue and processes them, don't make any assumptions about which instance of the service handles a specific message. Autoscaling could start additional instances of a service as the queue length grows. The Competing Consumers pattern describes how to handle this scenario.
- If the solution implements a long-running task, design this task to support both scaling out and scaling in. Without due care, such a task could prevent an instance of a process from being shut down cleanly when the system scales in, or it could lose data if the process is forcibly terminated. Ideally, refactor a long-running task and break up the processing that it performs into smaller, discrete chunks. The Pipes and Filters pattern provides an example of how you can achieve this.

- Alternatively, you can implement a checkpoint mechanism that records state information about the task at regular intervals, and save this state in durable storage that can be accessed by any instance of the process running the task. In this way, if the process is shut down, the work that it was performing can be resumed from the last checkpoint by using another instance.
- When background tasks run on separate compute instances, such as in worker roles of a cloud-services–hosted application, you may need to scale different parts of the application using different scaling policies. For example, you may need to deploy additional user interface (UI) compute instances without increasing the number of background compute instances, or the opposite of this. If you offer different levels of service (such as basic and premium service packages), you may need to scale out the compute resources for premium service packages more aggressively than those for basic service packages in order to meet SLAs.
- Consider using the length of the queue over which UI and background compute instances communicate as a criterion for your autoscaling strategy. This is the best indicator of an imbalance or difference between the current load and the processing capacity of the background task.
- If you base your autoscaling strategy on counters that measure business processes, such as the number of orders placed per hour or the average execution time of a complex transaction, ensure that you fully understand the relationship between the results from these types of counters and the actual compute capacity requirements. It may be necessary to scale more than one component or compute unit in response to changes in business process counters.
- To prevent a system from attempting to scale out excessively, and to avoid the costs associated with running many thousands of instances, consider limiting the maximum number of instances that can be automatically added. Most autoscaling mechanisms allow you to specify the minimum and maximum number of instances for a rule. In addition, consider gracefully degrading the functionality that the system provides if the maximum number of instances have been deployed, and the system is still overloaded.
- Keep in mind that autoscaling might not be the most appropriate mechanism to handle a sudden burst in workload. It takes time to provision and start new instances of a service or add resources to a system, and the peak demand may have passed by the time these additional resources have been made available. In this scenario, it may be better to throttle the service. For more information, see the Throttling pattern.
- Conversely, if you do need the capacity to process all requests when the volume fluctuates rapidly, and cost isn't a major contributing factor, consider using an aggressive autoscaling strategy that starts additional instances more quickly. You can also use a scheduled policy that starts a sufficient number of instances to meet the maximum load before that load is expected.
- The autoscaling mechanism should monitor the autoscaling process, and log the details of each autoscaling event (what triggered it, what resources were added or removed, and when). If you create a custom autoscaling mechanism, ensure that it incorporates this capability. Analyze the information to help measure the effectiveness of the autoscaling strategy, and tune it if necessary. You can tune both in the short term, as the usage patterns become more obvious, and over the long term, as the business expands or the requirements of the application evolve. If an application reaches the upper limit defined for autoscaling, the mechanism might also alert an operator who could manually start additional resources if necessary. Note that under these circumstances the operator may also be responsible for manually removing these resources after the workload eases.

Related patterns and guidance

The following patterns and guidance may also be relevant to your scenario when implementing autoscaling:

- Throttling pattern. This pattern describes how an application can continue to function and meet SLAs when an increase in demand places an extreme load on resources. Throttling can be used with autoscaling to prevent a system from being overwhelmed while the system scales out.

- Competing Consumers pattern. This pattern describes how to implement a pool of service instances that can handle messages from any application instance. Autoscaling can be used to start and stop service instances to match the anticipated workload. This approach enables a system to process multiple messages concurrently to optimize throughput, improve scalability and availability, and balance the workload.
- Monitoring and diagnostics. Instrumentation and telemetry are vital for gathering the information that can drive the autoscaling process.

Background jobs

11/19/2021 • 25 minutes to read • Edit Online

Many types of applications require background tasks that run independently of the user interface (UI). Examples include batch jobs, intensive processing tasks, and long-running processes such as workflows. Background jobs can be executed without requiring user interaction--the application can start the job and then continue to process interactive requests from users. This can help to minimize the load on the application UI, which can improve availability and reduce interactive response times.

For example, if an application is required to generate thumbnails of images that are uploaded by users, it can do this as a background job and save the thumbnail to storage when it is complete--without the user needing to wait for the process to be completed. In the same way, a user placing an order can initiate a background workflow that processes the order, while the UI allows the user to continue browsing the web app. When the background job is complete, it can update the stored orders data and send an email to the user that confirms the order.

When you consider whether to implement a task as a background job, the main criteria is whether the task can run without user interaction and without the UI needing to wait for the job to be completed. Tasks that require the user or the UI to wait while they are completed might not be appropriate as background jobs.

Types of background jobs

Background jobs typically include one or more of the following types of jobs:

- CPU-intensive jobs, such as mathematical calculations or structural model analysis.
- I/O-intensive jobs, such as executing a series of storage transactions or indexing files.
- Batch jobs, such as nightly data updates or scheduled processing.
- Long-running workflows, such as order fulfillment, or provisioning services and systems.
- Sensitive-data processing where the task is handed off to a more secure location for processing. For example, you might not want to process sensitive data within a web app. Instead, you might use a pattern such as the Gatekeeper pattern to transfer the data to an isolated background process that has access to protected storage.

Triggers

Background jobs can be initiated in several different ways. They fall into one of the following categories:

- **Event-driven triggers**. The task is started in response to an event, typically an action taken by a user or a step in a workflow.
- **Schedule-driven triggers**. The task is invoked on a schedule based on a timer. This might be a recurring schedule or a one-off invocation that is specified for a later time.

Event-driven triggers

Event-driven invocation uses a trigger to start the background task. Examples of using event-driven triggers include:

- The UI or another job places a message in a queue. The message contains data about an action that has taken place, such as the user placing an order. The background task listens on this queue and detects the arrival of a new message. It reads the message and uses the data in it as the input to the background job.
- The UI or another job saves or updates a value in storage. The background task monitors the storage and detects changes. It reads the data and uses it as the input to the background job.

- The UI or another job makes a request to an endpoint, such as an HTTPS URI, or an API that is exposed as a web service. It passes the data that is required to complete the background task as part of the request. The endpoint or web service invokes the background task, which uses the data as its input.

Typical examples of tasks that are suited to event-driven invocation include image processing, workflows, sending information to remote services, sending email messages, and provisioning new users in multitenant applications.

Schedule-driven triggers

Schedule-driven invocation uses a timer to start the background task. Examples of using schedule-driven triggers include:

- A timer that is running locally within the application or as part of the application's operating system invokes a background task on a regular basis.
- A timer that is running in a different application, such as Azure Logic Apps, sends a request to an API or web service on a regular basis. The API or web service invokes the background task.
- A separate process or application starts a timer that causes the background task to be invoked once after a specified time delay, or at a specific time.

Typical examples of tasks that are suited to schedule-driven invocation include batch-processing routines (such as updating related-products lists for users based on their recent behavior), routine data processing tasks (such as updating indexes or generating accumulated results), data analysis for daily reports, data retention cleanup, and data consistency checks.

If you use a schedule-driven task that must run as a single instance, be aware of the following:

- If the compute instance that is running the scheduler (such as a virtual machine using Windows scheduled tasks) is scaled, you will have multiple instances of the scheduler running. These could start multiple instances of the task.
- If tasks run for longer than the period between scheduler events, the scheduler may start another instance of the task while the previous one is still running.

Returning results

Background jobs execute asynchronously in a separate process, or even in a separate location, from the UI or the process that invoked the background task. Ideally, background tasks are "fire and forget" operations, and their execution progress has no impact on the UI or the calling process. This means that the calling process does not wait for completion of the tasks. Therefore, it cannot automatically detect when the task ends.

If you require a background task to communicate with the calling task to indicate progress or completion, you must implement a mechanism for this. Some examples are:

- Write a status indicator value to storage that is accessible to the UI or caller task, which can monitor or check this value when required. Other data that the background task must return to the caller can be placed into the same storage.
- Establish a reply queue that the UI or caller listens on. The background task can send messages to the queue that indicate status and completion. Data that the background task must return to the caller can be placed into the messages. If you are using Azure Service Bus, you can use the **ReplyTo** and **CorrelationId** properties to implement this capability.
- Expose an API or endpoint from the background task that the UI or caller can access to obtain status information. Data that the background task must return to the caller can be included in the response.
- Have the background task call back to the UI or caller through an API to indicate status at predefined points or on completion. This might be through events raised locally or through a publish-and-subscribe mechanism. Data that the background task must return to the caller can be included in the request or event

payload.

Hosting environment

You can host background tasks by using a range of different Azure platform services:

- **Azure Web Apps and WebJobs**. You can use WebJobs to execute custom jobs based on a range of different types of scripts or executable programs within the context of a web app.
- **Azure Functions**. You can use functions for background jobs that don't run for a long time. Another use case is if your workload is already hosted on App Service plan and is underutilized.
- **Azure Virtual Machines**. If you have a Windows service or want to use the Windows Task Scheduler, it is common to host your background tasks within a dedicated virtual machine.
- **Azure Batch**. Batch is a platform service that schedules compute-intensive work to run on a managed collection of virtual machines. It can automatically scale compute resources.
- **Azure Kubernetes Service** (AKS). Azure Kubernetes Service provides a managed hosting environment for Kubernetes on Azure.

The following sections describe each of these options in more detail, and include considerations to help you choose the appropriate option.

Azure Web Apps and WebJobs

You can use Azure WebJobs to execute custom jobs as background tasks within an Azure Web App. WebJobs run within the context of your web app as a continuous process. WebJobs also run in response to a trigger event from Azure Logic Apps or external factors, such as changes to storage blobs and message queues. Jobs can be started and stopped on demand, and shut down gracefully. If a continuously running WebJob fails, it is automatically restarted. Retry and error actions are configurable.

When you configure a WebJob:

- If you want the job to respond to an event-driven trigger, you should configure it as **Run continuously**. The script or program is stored in the folder named site/wwwroot/app_data/jobs/continuous.
- If you want the job to respond to a schedule-driven trigger, you should configure it as **Run on a schedule**. The script or program is stored in the folder named site/wwwroot/app_data/jobs/triggered.
- If you choose the **Run on demand** option when you configure a job, it will execute the same code as the **Run on a schedule** option when you start it.

Azure WebJobs run within the sandbox of the web app. This means that they can access environment variables and share information, such as connection strings, with the web app. The job has access to the unique identifier of the machine that is running the job. The connection string named **AzureWebJobsStorage** provides access to Azure storage queues, blobs, and tables for application data, and access to Service Bus for messaging and communication. The connection string named **AzureWebJobsDashboard** provides access to the job action log files.

Azure WebJobs have the following characteristics:

- **Security**: WebJobs are protected by the deployment credentials of the web app.
- **Supported file types**: You can define WebJobs by using command scripts (.cmd), batch files (.bat), PowerShell scripts (.ps1), bash shell scripts (.sh), PHP scripts (.php), Python scripts (.py), JavaScript code (.js), and executable programs (.exe, .jar, and more).
- **Deployment**: You can deploy scripts and executables by using the Azure portal, by using Visual Studio, by using the Azure WebJobs SDK, or by copying them directly to the following locations:
  - For triggered execution: site/wwwroot/app_data/jobs/triggered/{job name}
  - For continuous execution: site/wwwroot/app_data/jobs/continuous/{job name}
- **Logging**: Console.Out is treated (marked) as INFO. Console.Error is treated as ERROR. You can access

monitoring and diagnostics information by using the Azure portal. You can download log files directly from the site. They are saved in the following locations:
  - For triggered execution: Vfs/data/jobs/triggered/jobName
  - For continuous execution: Vfs/data/jobs/continuous/jobName
- **Configuration**: You can configure WebJobs by using the portal, the REST API, and PowerShell. You can use a configuration file named settings.job in the same root directory as the job script to provide configuration information for a job. For example:
  - { "stopping_wait_time": 60 }
  - { "is_singleton": true }

Considerations

- By default, WebJobs scale with the web app. However, you can configure jobs to run on single instance by setting the **is_singleton** configuration property to **true**. Single instance WebJobs are useful for tasks that you do not want to scale or run as simultaneous multiple instances, such as reindexing, data analysis, and similar tasks.
- To minimize the impact of jobs on the performance of the web app, consider creating an empty Azure Web App instance in a new App Service plan to host long-running or resource-intensive WebJobs.

Azure Functions

An option that is similar to WebJobs is Azure Functions. This service is serverless that is most suitable for event-driven triggers that run for a short period. A function can also be used to run scheduled jobs through timer triggers, when configured to run at set times.

Azure Functions is not a recommended option for large, long-running tasks because they can cause unexpected timeout issues. However, depending on the hosting plan, they can be considered for schedule-driven triggers.

Considerations

If the background task is expected to run for a short duration in response to an event, consider running the task in a Consumption plan. The execution time is configurable up to a maximum time. A function that runs for longer costs more. Also CPU-intensive jobs that consume more memory can be more expensive. If you use additional triggers for services as part of your task, those are billed separately.

The Premium plan is more suitable if you have a high number of tasks that are short but expected to run continuously. This plan is more expensive because it needs more memory and CPU. The benefit is that you can use features such as virtual network integration.

The Dedicated plan is most suitable for background jobs if your workload already runs on it. If you have underutilized VMs, you can run it on the same VM and share compute costs.

For more information, see these articles:

- Azure Functions hosting options
- Timer trigger for Azure Functions

Azure Virtual Machines

Background tasks might be implemented in a way that prevents them from being deployed to Azure Web Apps, or these options might not be convenient. Typical examples are Windows services, and third-party utilities and executable programs. Another example might be programs written for an execution environment that is different than that hosting the application. For example, it might be a Unix or Linux program that you want to execute from a Windows or .NET application. You can choose from a range of operating systems for an Azure virtual machine, and run your service or executable on that virtual machine.

To help you choose when to use Virtual Machines, see Azure App Services, Cloud Services and Virtual Machines comparison. For information about the options for Virtual Machines, see Sizes for Windows virtual machines in Azure. For more information about the operating systems and prebuilt images that are available for Virtual

Machines, see Azure Virtual Machines Marketplace.

To initiate the background task in a separate virtual machine, you have a range of options:

- You can execute the task on demand directly from your application by sending a request to an endpoint that the task exposes. This passes in any data that the task requires. This endpoint invokes the task.
- You can configure the task to run on a schedule by using a scheduler or timer that is available in your chosen operating system. For example, on Windows you can use Windows Task Scheduler to execute scripts and tasks. Or, if you have SQL Server installed on the virtual machine, you can use the SQL Server Agent to execute scripts and tasks.
- You can use Azure Logic Apps to initiate the task by adding a message to a queue that the task listens on, or by sending a request to an API that the task exposes.

See the earlier section Triggers for more information about how you can initiate background tasks.

Considerations

Consider the following points when you are deciding whether to deploy background tasks in an Azure virtual machine:

- Hosting background tasks in a separate Azure virtual machine provides flexibility and allows precise control over initiation, execution, scheduling, and resource allocation. However, it will increase runtime cost if a virtual machine must be deployed just to run background tasks.
- There is no facility to monitor the tasks in the Azure portal and no automated restart capability for failed tasks--although you can monitor the basic status of the virtual machine and manage it by using the Azure Resource Manager Cmdlets. However, there are no facilities to control processes and threads in compute nodes. Typically, using a virtual machine will require additional effort to implement a mechanism that collects data from instrumentation in the task, and from the operating system in the virtual machine. One solution that might be appropriate is to use the System Center Management Pack for Azure.
- You might consider creating monitoring probes that are exposed through HTTP endpoints. The code for these probes could perform health checks, collect operational information and statistics--or collate error information and return it to a management application. For more information, see the Health Endpoint Monitoring pattern.

For more information, see:

- Virtual Machines
- Azure Virtual Machines FAQ

Azure Batch

Consider Azure Batch if you need to run large, parallel high-performance computing (HPC) workloads across tens, hundreds, or thousands of VMs.

The Batch service provisions the VMs, assign tasks to the VMs, runs the tasks, and monitors the progress. Batch can automatically scale out the VMs in response to the workload. Batch also provides job scheduling. Azure Batch supports both Linux and Windows VMs.

Considerations

Batch works well with intrinsically parallel workloads. It can also perform parallel calculations with a reduce step at the end, or run Message Passing Interface (MPI) applications for parallel tasks that require message passing between nodes.

An Azure Batch job runs on a pool of nodes (VMs). One approach is to allocate a pool only when needed and then delete it after the job completes. This maximizes utilization, because nodes are not idle, but the job must wait for nodes to be allocated. Alternatively, you can create a pool ahead of time. That approach minimizes the time that it takes for a job to start, but can result in having nodes that sit idle. For more information, see Pool and compute node lifetime.

For more information, see:

- What is Azure Batch?
- Develop large-scale parallel compute solutions with Batch
- Batch and HPC solutions for large-scale computing workloads

Azure Kubernetes Service

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, which makes it easy to deploy and manage containerized applications.

Containers can be useful for running background jobs. Some of the benefits include:

- Containers support high-density hosting. You can isolate a background task in a container, while placing multiple containers in each VM.
- The container orchestrator handles internal load balancing, configuring the internal network, and other configuration tasks.
- Containers can be started and stopped as needed.
- Azure Container Registry allows you to register your containers inside Azure boundaries. This comes with security, privacy, and proximity benefits.

Considerations

- Requires an understanding of how to use a container orchestrator. Depending on the skill set of your DevOps team, this may or may not be an issue.

For more information, see:

- Overview of containers in Azure
- Introduction to private Docker container registries

Partitioning

If you decide to include background tasks within an existing compute instance, you must consider how this will affect the quality attributes of the compute instance and the background task itself. These factors will help you to decide whether to colocate the tasks with the existing compute instance or separate them out into a separate compute instance:

- **Availability**: Background tasks might not need to have the same level of availability as other parts of the application, in particular the UI and other parts that are directly involved in user interaction. Background tasks might be more tolerant of latency, retried connection failures, and other factors that affect availability because the operations can be queued. However, there must be sufficient capacity to prevent the backup of requests that could block queues and affect the application as a whole.
- **Scalability**: Background tasks are likely to have a different scalability requirement than the UI and the interactive parts of the application. Scaling the UI might be necessary to meet peaks in demand, while outstanding background tasks might be completed during less busy times by fewer compute instances.
- **Resiliency**: Failure of a compute instance that just hosts background tasks might not fatally affect the application as a whole if the requests for these tasks can be queued or postponed until the task is available again. If the compute instance and/or tasks can be restarted within an appropriate interval, users of the application might not be affected.
- **Security**: Background tasks might have different security requirements or restrictions than the UI or other parts of the application. By using a separate compute instance, you can specify a different security environment for the tasks. You can also use patterns such as Gatekeeper to isolate the background compute instances from the UI in order to maximize security and separation.

- **Performance**: You can choose the type of compute instance for background tasks to specifically match the performance requirements of the tasks. This might mean using a less expensive compute option if the tasks do not require the same processing capabilities as the UI, or a larger instance if they require additional capacity and resources.
- **Manageability**: Background tasks might have a different development and deployment rhythm from the main application code or the UI. Deploying them to a separate compute instance can simplify updates and versioning.
- **Cost**: Adding compute instances to execute background tasks increases hosting costs. You should carefully consider the trade-off between additional capacity and these extra costs.

For more information, see the Leader Election pattern and the Competing Consumers pattern.

Conflicts

If you have multiple instances of a background job, it is possible that they will compete for access to resources and services, such as databases and storage. This concurrent access can result in resource contention, which might cause conflicts in availability of the services and in the integrity of data in storage. You can resolve resource contention by using a pessimistic locking approach. This prevents competing instances of a task from concurrently accessing a service or corrupting data.

Another approach to resolve conflicts is to define background tasks as a singleton, so that there is only ever one instance running. However, this eliminates the reliability and performance benefits that a multiple-instance configuration can provide. This is especially true if the UI can supply sufficient work to keep more than one background task busy.

It is vital to ensure that the background task can automatically restart and that it has sufficient capacity to cope with peaks in demand. You can achieve this by allocating a compute instance with sufficient resources, by implementing a queueing mechanism that can store requests for later execution when demand decreases, or by using a combination of these techniques.

Coordination

The background tasks might be complex and might require multiple individual tasks to execute to produce a result or to fulfill all the requirements. It is common in these scenarios to divide the task into smaller discreet steps or subtasks that can be executed by multiple consumers. Multistep jobs can be more efficient and more flexible because individual steps might be reusable in multiple jobs. It is also easy to add, remove, or modify the order of the steps.

Coordinating multiple tasks and steps can be challenging, but there are three common patterns that you can use to guide your implementation of a solution:

- **Decomposing a task into multiple reusable steps**. An application might be required to perform a variety of tasks of varying complexity on the information that it processes. A straightforward but inflexible approach to implementing this application might be to perform this processing as a monolithic module. However, this approach is likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing are required elsewhere within the application. For more information, see the Pipes and Filters pattern.
- **Managing execution of the steps for a task**. An application might perform tasks that comprise a number of steps (some of which might invoke remote services or access remote resources). The individual steps might be independent of each other, but they are orchestrated by the application logic that implements the task. For more information, see Scheduler Agent Supervisor pattern.
- **Managing recovery for task steps that fail**. An application might need to undo the work that is

performed by a series of steps (which together define an eventually consistent operation) if one or more of the steps fail. For more information, see the Compensating Transaction pattern.

Resiliency considerations

Background tasks must be resilient in order to provide reliable services to the application. When you are planning and designing background tasks, consider the following points:

- Background tasks must be able to gracefully handle restarts without corrupting data or introducing inconsistency into the application. For long-running or multistep tasks, consider using *check pointing* by saving the state of jobs in persistent storage, or as messages in a queue if this is appropriate. For example, you can persist state information in a message in a queue and incrementally update this state information with the task progress so that the task can be processed from the last known good checkpoint--instead of restarting from the beginning. When using Azure Service Bus queues, you can use message sessions to enable the same scenario. Sessions allow you to save and retrieve the application processing state by using the SetState and GetState methods. For more information about designing reliable multistep processes and workflows, see the Scheduler Agent Supervisor pattern.
- When you use queues to communicate with background tasks, the queues can act as a buffer to store requests that are sent to the tasks while the application is under higher than usual load. This allows the tasks to catch up with the UI during less busy periods. It also means that restarts will not block the UI. For more information, see the Queue-Based Load Leveling pattern. If some tasks are more important than others, consider implementing the Priority Queue pattern to ensure that these tasks run before less important ones.
- Background tasks that are initiated by messages or process messages must be designed to handle inconsistencies, such as messages arriving out of order, messages that repeatedly cause an error (often referred to as *poison messages*), and messages that are delivered more than once. Consider the following:
    - Messages that must be processed in a specific order, such as those that change data based on the existing data value (for example, adding a value to an existing value), might not arrive in the original order in which they were sent. Alternatively, they might be handled by different instances of a background task in a different order due to varying loads on each instance. Messages that must be processed in a specific order should include a sequence number, key, or some other indicator that background tasks can use to ensure that they are processed in the correct order. If you are using Azure Service Bus, you can use message sessions to guarantee the order of delivery. However, it is usually more efficient, where possible, to design the process so that the message order is not important.
    - Typically, a background task will peek at messages in the queue, which temporarily hides them from other message consumers. Then it deletes the messages after they have been successfully processed. If a background task fails when processing a message, that message will reappear on the queue after the peek time-out expires. It will be processed by another instance of the task or during the next processing cycle of this instance. If the message consistently causes an error in the consumer, it will block the task, the queue, and eventually the application itself when the queue becomes full. Therefore, it is vital to detect and remove poison messages from the queue. If you are using Azure Service Bus, messages that cause an error can be moved automatically or manually to an associated dead letter queue.
    - Queues are guaranteed at *least once* delivery mechanisms, but they might deliver the same message more than once. In addition, if a background task fails after processing a message but before deleting it from the queue, the message will become available for processing again. Background tasks should be idempotent, which means that processing the same message more than once does not cause an error or inconsistency in the application's data. Some operations are

naturally idempotent, such as setting a stored value to a specific new value. However, operations such as adding a value to an existing stored value without checking that the stored value is still the same as when the message was originally sent will cause inconsistencies. Azure Service Bus queues can be configured to automatically remove duplicated messages.

- Some messaging systems, such as Azure storage queues and Azure Service Bus queues, support a de-queue count property that indicates the number of times a message has been read from the queue. This can be useful in handling repeated and poison messages. For more information, see Asynchronous Messaging Primer and Idempotency Patterns.

Scaling and performance considerations

Background tasks must offer sufficient performance to ensure they do not block the application, or cause inconsistencies due to delayed operation when the system is under load. Typically, performance is improved by scaling the compute instances that host the background tasks. When you are planning and designing background tasks, consider the following points around scalability and performance:

- Azure supports autoscaling (both scaling out and scaling back in) based on current demand and load or on a predefined schedule, for Web Apps and Virtual Machines hosted deployments. Use this feature to ensure that the application as a whole has sufficient performance capabilities while minimizing runtime costs.
- Where background tasks have a different performance capability from the other parts of an application (for example, the UI or components such as the data access layer), hosting the background tasks together in a separate compute service allows the UI and background tasks to scale independently to manage the load. If multiple background tasks have significantly different performance capabilities from each other, consider dividing them and scaling each type independently. However, note that this might increase runtime costs.
- Simply scaling the compute resources might not be sufficient to prevent loss of performance under load. You might also need to scale storage queues and other resources to prevent a single point of the overall processing chain from becoming a bottleneck. Also, consider other limitations, such as the maximum throughput of storage and other services that the application and the background tasks rely on.
- Background tasks must be designed for scaling. For example, they must be able to dynamically detect the number of storage queues in use in order to listen on or send messages to the appropriate queue.
- By default, WebJobs scale with their associated Azure Web Apps instance. However, if you want a WebJob to run as only a single instance, you can create a Settings.job file that contains the JSON data `{ "is_singleton": true }`. This forces Azure to only run one instance of the WebJob, even if there are multiple instances of the associated web app. This can be a useful technique for scheduled jobs that must run as only a single instance.

Related patterns

- Compute Partitioning Guidance

Caching

11/19/2021 • 55 minutes to read • Edit Online

Caching is a common technique that aims to improve the performance and scalability of a system. It does this by temporarily copying frequently accessed data to fast storage that's located close to the application. If this fast data storage is located closer to the application than the original source, then caching can significantly improve response times for client applications by serving data more quickly.

Caching is most effective when a client instance repeatedly reads the same data, especially if all the following conditions apply to the original data store:

- It remains relatively static.
- It's slow compared to the speed of the cache.
- It's subject to a high level of contention.
- It's far away when network latency can cause access to be slow.

Caching in distributed applications

Distributed applications typically implement either or both of the following strategies when caching data:

- Using a private cache, where data is held locally on the computer that's running an instance of an application or service.
- Using a shared cache, serving as a common source that can be accessed by multiple processes and machines.

In both cases, caching can be performed client-side and server-side. Client-side caching is done by the process that provides the user interface for a system, such as a web browser or desktop application. Server-side caching is done by the process that provides the business services that are running remotely.

Private caching

The most basic type of cache is an in-memory store. It's held in the address space of a single process and accessed directly by the code that runs in that process. This type of cache is quick to access. It can also provide an effective means for storing modest amounts of static data, since the size of a cache is typically constrained by the amount of memory available on the machine hosting the process.

If you need to cache more information than is physically possible in memory, you can write cached data to the local file system. This will be slower to access than data held in memory, but should still be faster and more reliable than retrieving data across a network.

If you have multiple instances of an application that uses this model running concurrently, each application instance has its own independent cache holding its own copy of the data.

Think of a cache as a snapshot of the original data at some point in the past. If this data is not static, it is likely that different application instances hold different versions of the data in their caches. Therefore, the same query performed by these instances can return different results, as shown in Figure 1.

Figure 1: Using an in-memory cache in different instances of an application.

Shared caching

Using a shared cache can help alleviate concerns that data might differ in each cache, which can occur with in-memory caching. Shared caching ensures that different application instances see the same view of cached data. It does this by locating the cache in a separate location, typically hosted as part of a separate service, as shown in Figure 2.

Figure 2: Using a shared cache.

An important benefit of the shared caching approach is the scalability it provides. Many shared cache services are implemented by using a cluster of servers and use software to distribute the data across the cluster transparently. An application instance simply sends a request to the cache service. The underlying infrastructure

determines the location of the cached data in the cluster. You can easily scale the cache by adding more servers.

There are two main disadvantages of the shared caching approach:

- The cache is slower to access because it is no longer held locally to each application instance.
- The requirement to implement a separate cache service might add complexity to the solution.

Considerations for using caching

The following sections describe in more detail the considerations for designing and using a cache.

Decide when to cache data

Caching can dramatically improve performance, scalability, and availability. The more data that you have and the larger the number of users that need to access this data, the greater the benefits of caching become. That's because caching reduces the latency and contention that's associated with handling large volumes of concurrent requests in the original data store.

For example, a database might support a limited number of concurrent connections. Retrieving data from a shared cache, however, rather than the underlying database, makes it possible for a client application to access this data even if the number of available connections is currently exhausted. Additionally, if the database becomes unavailable, client applications might be able to continue by using the data that's held in the cache.

Consider caching data that is read frequently but modified infrequently (for example, data that has a higher proportion of read operations than write operations). However, we don't recommend that you use the cache as the authoritative store of critical information. Instead, ensure that all changes that your application cannot afford to lose are always saved to a persistent data store. This means that if the cache is unavailable, your application can still continue to operate by using the data store, and you won't lose important information.

Determine how to cache data effectively

The key to using a cache effectively lies in determining the most appropriate data to cache, and caching it at the appropriate time. The data can be added to the cache on demand the first time it is retrieved by an application. This means that the application needs to fetch the data only once from the data store, and that subsequent access can be satisfied by using the cache.

Alternatively, a cache can be partially or fully populated with data in advance, typically when the application starts (an approach known as seeding). However, it might not be advisable to implement seeding for a large cache because this approach can impose a sudden, high load on the original data store when the application starts running.

Often an analysis of usage patterns can help you decide whether to fully or partially prepopulate a cache, and to choose the data to cache. For example, it can be useful to seed the cache with the static user profile data for customers who use the application regularly (perhaps every day), but not for customers who use the application only once a week.

Caching typically works well with data that is immutable or that changes infrequently. Examples include reference information such as product and pricing information in an e-commerce application, or shared static resources that are costly to construct. Some or all of this data can be loaded into the cache at application startup to minimize demand on resources and to improve performance. It might also be appropriate to have a background process that periodically updates reference data in the cache to ensure it is up-to-date, or that refreshes the cache when reference data changes.

Caching is less useful for dynamic data, although there are some exceptions to this consideration (see the section Cache highly dynamic data later in this article for more information). When the original data changes regularly, either the cached information becomes stale very quickly or the overhead of synchronizing the cache with the original data store reduces the effectiveness of caching.

Note that a cache does not have to include the complete data for an entity. For example, if a data item represents

a multivalued object such as a bank customer with a name, address, and account balance, some of these elements might remain static (such as the name and address), while others (such as the account balance) might be more dynamic. In these situations, it can be useful to cache the static portions of the data and retrieve (or calculate) only the remaining information when it is required.

We recommend that you carry out performance testing and usage analysis to determine whether prepopulating or on-demand loading of the cache, or a combination of both, is appropriate. The decision should be based on the volatility and usage pattern of the data. Cache utilization and performance analysis are particularly important in applications that encounter heavy loads and must be highly scalable. For example, in highly scalable scenarios it might make sense to seed the cache to reduce the load on the data store at peak times.

Caching can also be used to avoid repeating computations while the application is running. If an operation transforms data or performs a complicated calculation, it can save the results of the operation in the cache. If the same calculation is required afterward, the application can simply retrieve the results from the cache.

An application can modify data that's held in a cache. However, we recommend thinking of the cache as a transient data store that could disappear at any time. Do not store valuable data in the cache only; make sure that you maintain the information in the original data store as well. This means that if the cache becomes unavailable, you minimize the chance of losing data.

Cache highly dynamic data

When you store rapidly changing information in a persistent data store, it can impose an overhead on the system. For example, consider a device that continually reports status or some other measurement. If an application chooses not to cache this data on the basis that the cached information will nearly always be outdated, then the same consideration could be true when storing and retrieving this information from the data store. In the time it takes to save and fetch this data, it might have changed.

In a situation such as this, consider the benefits of storing the dynamic information directly in the cache instead of in the persistent data store. If the data is noncritical and does not require auditing, then it doesn't matter if the occasional change is lost.

Manage data expiration in a cache

In most cases, data that's held in a cache is a copy of data that's held in the original data store. The data in the original data store might change after it was cached, causing the cached data to become stale. Many caching systems enable you to configure the cache to expire data and reduce the period for which data may be out of date.

When cached data expires, it's removed from the cache, and the application must retrieve the data from the original data store (it can put the newly fetched information back into cache). You can set a default expiration policy when you configure the cache. In many cache services, you can also stipulate the expiration period for individual objects when you store them programmatically in the cache. Some caches enable you to specify the expiration period as an absolute value, or as a sliding value that causes the item to be removed from the cache if it is not accessed within the specified time. This setting overrides any cache-wide expiration policy, but only for the specified objects.

> **NOTE**
>
> Consider the expiration period for the cache and the objects that it contains carefully. If you make it too short, objects will expire too quickly and you will reduce the benefits of using the cache. If you make the period too long, you risk the data becoming stale.

It's also possible that the cache might fill up if data is allowed to remain resident for a long time. In this case, any requests to add new items to the cache might cause some items to be forcibly removed in a process known as eviction. Cache services typically evict data on a least-recently-used (LRU) basis, but you can usually override this policy and prevent items from being evicted. However, if you adopt this approach, you risk exceeding the

memory that's available in the cache. An application that attempts to add an item to the cache will fail with an exception.

Some caching implementations might provide additional eviction policies. There are several types of eviction policies. These include:

- A most-recently-used policy (in the expectation that the data will not be required again).
- A first-in-first-out policy (oldest data is evicted first).
- An explicit removal policy based on a triggered event (such as the data being modified).

Invalidate data in a client-side cache

Data that's held in a client-side cache is generally considered to be outside the auspices of the service that provides the data to the client. A service cannot directly force a client to add or remove information from a client-side cache.

This means that it's possible for a client that uses a poorly configured cache to continue using outdated information. For example, if the expiration policies of the cache aren't properly implemented, a client might use outdated information that's cached locally when the information in the original data source has changed.

If you are building a web application that serves data over an HTTP connection, you can implicitly force a web client (such as a browser or web proxy) to fetch the most recent information. You can do this if a resource is updated by a change in the URI of that resource. Web clients typically use the URI of a resource as the key in the client-side cache, so if the URI changes, the web client ignores any previously cached versions of a resource and fetches the new version instead.

Managing concurrency in a cache

Caches are often designed to be shared by multiple instances of an application. Each application instance can read and modify data in the cache. Consequently, the same concurrency issues that arise with any shared data store also apply to a cache. In a situation where an application needs to modify data that's held in the cache, you might need to ensure that updates made by one instance of the application do not overwrite the changes made by another instance.

Depending on the nature of the data and the likelihood of collisions, you can adopt one of two approaches to concurrency:

- **Optimistic**. Immediately prior to updating the data, the application checks to see whether the data in the cache has changed since it was retrieved. If the data is still the same, the change can be made. Otherwise, the application has to decide whether to update it. (The business logic that drives this decision will be application-specific.) This approach is suitable for situations where updates are infrequent, or where collisions are unlikely to occur.
- **Pessimistic**. When it retrieves the data, the application locks it in the cache to prevent another instance from changing it. This process ensures that collisions cannot occur, but they can also block other instances that need to process the same data. Pessimistic concurrency can affect the scalability of a solution and is recommended only for short-lived operations. This approach might be appropriate for situations where collisions are more likely, especially if an application updates multiple items in the cache and must ensure that these changes are applied consistently.

Implement high availability and scalability, and improve performance

Avoid using a cache as the primary repository of data; this is the role of the original data store from which the cache is populated. The original data store is responsible for ensuring the persistence of the data.

Be careful not to introduce critical dependencies on the availability of a shared cache service into your solutions. An application should be able to continue functioning if the service that provides the shared cache is unavailable. The application should not become unresponsive or fail while waiting for the cache service to

resume.

Therefore, the application must be prepared to detect the availability of the cache service and fall back to the original data store if the cache is inaccessible. The Circuit-Breaker pattern is useful for handling this scenario. The service that provides the cache can be recovered, and once it becomes available, the cache can be repopulated as data is read from the original data store, following a strategy such as the Cache-aside pattern.

However, system scalability may be affected if the application falls back to the original data store when the cache is temporarily unavailable. While the data store is being recovered, the original data store could be swamped with requests for data, resulting in timeouts and failed connections.

Consider implementing a local, private cache in each instance of an application, together with the shared cache that all application instances access. When the application retrieves an item, it can check first in its local cache, then in the shared cache, and finally in the original data store. The local cache can be populated using the data in either the shared cache, or in the database if the shared cache is unavailable.

This approach requires careful configuration to prevent the local cache from becoming too stale with respect to the shared cache. However, the local cache acts as a buffer if the shared cache is unreachable. Figure 3 shows this structure.

Figure 3: Using a local private cache with a shared cache.

To support large caches that hold relatively long-lived data, some cache services provide a high-availability option that implements automatic failover if the cache becomes unavailable. This approach typically involves replicating the cached data that's stored on a primary cache server to a secondary cache server, and switching to the secondary server if the primary server fails or connectivity is lost.

To reduce the latency that's associated with writing to multiple destinations, the replication to the secondary server might occur asynchronously when data is written to the cache on the primary server. This approach leads to the possibility that some cached information might be lost in the event of a failure, but the proportion of this data should be small compared to the overall size of the cache.

If a shared cache is large, it might be beneficial to partition the cached data across nodes to reduce the chances of contention and improve scalability. Many shared caches support the ability to dynamically add (and remove) nodes and rebalance the data across partitions. This approach might involve clustering, in which the collection

of nodes is presented to client applications as a seamless, single cache. Internally, however, the data is dispersed between nodes following a predefined distribution strategy that balances the load evenly. For more information about possible partitioning strategies, see Data partitioning guidance.

Clustering can also increase the availability of the cache. If a node fails, the remainder of the cache is still accessible. Clustering is frequently used in conjunction with replication and failover. Each node can be replicated, and the replica can be quickly brought online if the node fails.

Many read and write operations are likely to involve single data values or objects. However, at times it might be necessary to store or retrieve large volumes of data quickly. For example, seeding a cache could involve writing hundreds or thousands of items to the cache. An application might also need to retrieve a large number of related items from the cache as part of the same request.

Many large-scale caches provide batch operations for these purposes. This enables a client application to package up a large volume of items into a single request and reduces the overhead that's associated with performing a large number of small requests.

Caching and eventual consistency

For the cache-aside pattern to work, the instance of the application that populates the cache must have access to the most recent and consistent version of the data. In a system that implements eventual consistency (such as a replicated data store) this might not be the case.

One instance of an application could modify a data item and invalidate the cached version of that item. Another instance of the application might attempt to read this item from a cache, which causes a cache-miss, so it reads the data from the data store and adds it to the cache. However, if the data store has not been fully synchronized with the other replicas, the application instance could read and populate the cache with the old value.

For more information about handling data consistency, see the Data consistency primer.

Protect cached data

Irrespective of the cache service you use, consider how to protect the data that's held in the cache from unauthorized access. There are two main concerns:

- The privacy of the data in the cache.
- The privacy of data as it flows between the cache and the application that's using the cache.

To protect data in the cache, the cache service might implement an authentication mechanism that requires that applications specify the following:

- Which identities can access data in the cache.
- Which operations (read and write) that these identities are allowed to perform.

To reduce overhead that's associated with reading and writing data, after an identity has been granted write and/or read access to the cache, that identity can use any data in the cache.

If you need to restrict access to subsets of the cached data, you can do one of the following:

- Split the cache into partitions (by using different cache servers) and only grant access to identities for the partitions that they should be allowed to use.
- Encrypt the data in each subset by using different keys, and provide the encryption keys only to identities that should have access to each subset. A client application might still be able to retrieve all of the data in the cache, but it will only be able to decrypt the data for which it has the keys.

You must also protect the data as it flows in and out of the cache. To do this, you depend on the security features provided by the network infrastructure that client applications use to connect to the cache. If the cache is implemented using an on-site server within the same organization that hosts the client applications, then the

isolation of the network itself might not require you to take additional steps. If the cache is located remotely and requires a TCP or HTTP connection over a public network (such as the Internet), consider implementing SSL.

Considerations for implementing caching in Azure

Azure Cache for Redis is an implementation of the open source Redis cache that runs as a service in an Azure datacenter. It provides a caching service that can be accessed from any Azure application, whether the application is implemented as a cloud service, a website, or inside an Azure virtual machine. Caches can be shared by client applications that have the appropriate access key.

Azure Cache for Redis is a high-performance caching solution that provides availability, scalability and security. It typically runs as a service spread across one or more dedicated machines. It attempts to store as much information as it can in memory to ensure fast access. This architecture is intended to provide low latency and high throughput by reducing the need to perform slow I/O operations.

Azure Cache for Redis is compatible with many of the various APIs that are used by client applications. If you have existing applications that already use Azure Cache for Redis running on-premises, the Azure Cache for Redis provides a quick migration path to caching in the cloud.

Features of Redis

Redis is more than a simple cache server. It provides a distributed in-memory database with an extensive command set that supports many common scenarios. These are described later in this document, in the section Using Redis caching. This section summarizes some of the key features that Redis provides.

Redis as an in-memory database

Redis supports both read and write operations. In Redis, writes can be protected from system failure either by being stored periodically in a local snapshot file or in an append-only log file. This is not the case in many caches (which should be considered transitory data stores).

All writes are asynchronous and do not block clients from reading and writing data. When Redis starts running, it reads the data from the snapshot or log file and uses it to construct the in-memory cache. For more information, see Redis persistence on the Redis website.

> **NOTE**
> Redis does not guarantee that all writes will be saved in the event of a catastrophic failure, but at worst you might lose only a few seconds worth of data. Remember that a cache is not intended to act as an authoritative data source, and it is the responsibility of the applications using the cache to ensure that critical data is saved successfully to an appropriate data store. For more information, see the Cache-aside pattern.

Redis data types

Redis is a key-value store, where values can contain simple types or complex data structures such as hashes, lists, and sets. It supports a set of atomic operations on these data types. Keys can be permanent or tagged with a limited time-to-live, at which point the key and its corresponding value are automatically removed from the cache. For more information about Redis keys and values, visit the page An introduction to Redis data types and abstractions on the Redis website.

Redis replication and clustering

Redis supports primary/subordinate replication to help ensure availability and maintain throughput. Write operations to a Redis primary node are replicated to one or more subordinate nodes. Read operations can be served by the primary or any of the subordinates.

In the event of a network partition, subordinates can continue to serve data and then transparently resynchronize with the primary when the connection is reestablished. For further details, visit the Replication page on the Redis website.

Redis also provides clustering, which enables you to transparently partition data into shards across servers and spread the load. This feature improves scalability, because new Redis servers can be added and the data repartitioned as the size of the cache increases.

Furthermore, each server in the cluster can be replicated by using primary/subordinate replication. This ensures availability across each node in the cluster. For more information about clustering and sharding, visit the Redis cluster tutorial page on the Redis website.

Redis memory use

A Redis cache has a finite size that depends on the resources available on the host computer. When you configure a Redis server, you can specify the maximum amount of memory it can use. You can also configure a key in a Redis cache to have an expiration time, after which it is automatically removed from the cache. This feature can help prevent the in-memory cache from filling with old or stale data.

As memory fills up, Redis can automatically evict keys and their values by following a number of policies. The default is LRU (least recently used), but you can also select other policies such as evicting keys at random or turning off eviction altogether (in which, case attempts to add items to the cache fail if it is full). The page Using Redis as an LRU cache provides more information.

Redis transactions and batches

Redis enables a client application to submit a series of operations that read and write data in the cache as an atomic transaction. All the commands in the transaction are guaranteed to run sequentially, and no commands issued by other concurrent clients will be interwoven between them.

However, these are not true transactions as a relational database would perform them. Transaction processing consists of two stages--the first is when the commands are queued, and the second is when the commands are run. During the command queuing stage, the commands that comprise the transaction are submitted by the client. If some sort of error occurs at this point (such as a syntax error, or the wrong number of parameters) then Redis refuses to process the entire transaction and discards it.

During the run phase, Redis performs each queued command in sequence. If a command fails during this phase, Redis continues with the next queued command and does not roll back the effects of any commands that have already been run. This simplified form of transaction helps to maintain performance and avoid performance problems that are caused by contention.

Redis does implement a form of optimistic locking to assist in maintaining consistency. For detailed information about transactions and locking with Redis, visit the Transactions page on the Redis website.

Redis also supports nontransactional batching of requests. The Redis protocol that clients use to send commands to a Redis server enables a client to send a series of operations as part of the same request. This can help to reduce packet fragmentation on the network. When the batch is processed, each command is performed. If any of these commands are malformed, they will be rejected (which doesn't happen with a transaction), but the remaining commands will be performed. There is also no guarantee about the order in which the commands in the batch will be processed.

Redis security

Redis is focused purely on providing fast access to data, and is designed to run inside a trusted environment that can be accessed only by trusted clients. Redis supports a limited security model based on password authentication. (It is possible to remove authentication completely, although we don't recommend this.)

All authenticated clients share the same global password and have access to the same resources. If you need more comprehensive sign-in security, you must implement your own security layer in front of the Redis server, and all client requests should pass through this additional layer. Redis should not be directly exposed to untrusted or unauthenticated clients.

You can restrict access to commands by disabling them or renaming them (and by providing only privileged

clients with the new names).

Redis does not directly support any form of data encryption, so all encoding must be performed by client applications. Additionally, Redis does not provide any form of transport security. If you need to protect data as it flows across the network, we recommend implementing an SSL proxy.

For more information, visit the Redis security page on the Redis website.

> **NOTE**
> Azure Cache for Redis provides its own security layer through which clients connect. The underlying Redis servers are not exposed to the public network.

Azure Redis cache

Azure Cache for Redis provides access to Redis servers that are hosted at an Azure datacenter. It acts as a façade that provides access control and security. You can provision a cache by using the Azure portal.

The portal provides a number of predefined configurations. These range from a 53 GB cache running as a dedicated service that supports SSL communications (for privacy) and master/subordinate replication with an SLA of 99.9% availability, down to a 250 MB cache without replication (no availability guarantees) running on shared hardware.

Using the Azure portal, you can also configure the eviction policy of the cache, and control access to the cache by adding users to the roles provided. These roles, which define the operations that members can perform, include Owner, Contributor, and Reader. For example, members of the Owner role have complete control over the cache (including security) and its contents, members of the Contributor role can read and write information in the cache, and members of the Reader role can only retrieve data from the cache.

Most administrative tasks are performed through the Azure portal. For this reason, many of the administrative commands that are available in the standard version of Redis are not available, including the ability to modify the configuration programmatically, shut down the Redis server, configure additional subordinates, or forcibly save data to disk.

The Azure portal includes a convenient graphical display that enables you to monitor the performance of the cache. For example, you can view the number of connections being made, the number of requests being performed, the volume of reads and writes, and the number of cache hits versus cache misses. Using this information, you can determine the effectiveness of the cache and if necessary, switch to a different configuration or change the eviction policy.

Additionally, you can create alerts that send email messages to an administrator if one or more critical metrics fall outside of an expected range. For example, you might want to alert an administrator if the number of cache misses exceeds a specified value in the last hour, because it means the cache might be too small or data might be being evicted too quickly.

You can also monitor the CPU, memory, and network usage for the cache.

For further information and examples showing how to create and configure an Azure Cache for Redis, visit the page Lap around Azure Cache for Redis on the Azure blog.

Caching session state and HTML output

If you're building ASP.NET web applications that run by using Azure web roles, you can save session state information and HTML output in an Azure Cache for Redis. The session state provider for Azure Cache for Redis enables you to share session information between different instances of an ASP.NET web application, and is very useful in web farm situations where client-server affinity is not available and caching session data in-memory would not be appropriate.

Using the session state provider with Azure Cache for Redis delivers several benefits, including:

- Sharing session state with a large number of instances of ASP.NET web applications.
- Providing improved scalability.
- Supporting controlled, concurrent access to the same session state data for multiple readers and a single writer.
- Using compression to save memory and improve network performance.

For more information, see ASP.NET session state provider for Azure Cache for Redis.

> **NOTE**
> Do not use the session state provider for Azure Cache for Redis with ASP.NET applications that run outside of the Azure environment. The latency of accessing the cache from outside of Azure can eliminate the performance benefits of caching data.

Similarly, the output cache provider for Azure Cache for Redis enables you to save the HTTP responses generated by an ASP.NET web application. Using the output cache provider with Azure Cache for Redis can improve the response times of applications that render complex HTML output. Application instances that generate similar responses can use the shared output fragments in the cache rather than generating this HTML output afresh. For more information, see ASP.NET output cache provider for Azure Cache for Redis.

Building a custom Redis cache

Azure Cache for Redis acts as a façade to the underlying Redis servers. If you require an advanced configuration that is not covered by the Azure Redis cache (such as a cache bigger than 53 GB) you can build and host your own Redis servers by using Azure virtual machines.

This is a potentially complex process because you might need to create several VMs to act as primary and subordinate nodes if you want to implement replication. Furthermore, if you wish to create a cluster, then you need multiple primaries and subordinate servers. A minimal clustered replication topology that provides a high degree of availability and scalability comprises at least six VMs organized as three pairs of primary/subordinate servers (a cluster must contain at least three primary nodes).

Each primary/subordinate pair should be located close together to minimize latency. However, each set of pairs can be running in different Azure datacenters located in different regions, if you wish to locate cached data close to the applications that are most likely to use it. For an example of building and configuring a Redis node running as an Azure VM, see Running Redis on a CentOS Linux VM in Azure.

> **NOTE**
> If you implement your own Redis cache in this way, you are responsible for monitoring, managing, and securing the service.

Partitioning a Redis cache

Partitioning the cache involves splitting the cache across multiple computers. This structure gives you several advantages over using a single cache server, including:

- Creating a cache that is much bigger than can be stored on a single server.
- Distributing data across servers, improving availability. If one server fails or becomes inaccessible, the data that it holds is unavailable, but the data on the remaining servers can still be accessed. For a cache, this is not crucial because the cached data is only a transient copy of the data that's held in a database. Cached data on a server that becomes inaccessible can be cached on a different server instead.

- Spreading the load across servers, thereby improving performance and scalability.
- Geolocating data close to the users that access it, thus reducing latency.

For a cache, the most common form of partitioning is sharding. In this strategy, each partition (or shard) is a Redis cache in its own right. Data is directed to a specific partition by using sharding logic, which can use a variety of approaches to distribute the data. The Sharding pattern provides more information about implementing sharding.

To implement partitioning in a Redis cache, you can take one of the following approaches:

- *Server-side query routing.* In this technique, a client application sends a request to any of the Redis servers that comprise the cache (probably the closest server). Each Redis server stores metadata that describes the partition that it holds, and also contains information about which partitions are located on other servers. The Redis server examines the client request. If it can be resolved locally, it will perform the requested operation. Otherwise it will forward the request on to the appropriate server. This model is implemented by Redis clustering, and is described in more detail on the Redis cluster tutorial page on the Redis website. Redis clustering is transparent to client applications, and additional Redis servers can be added to the cluster (and the data re-partitioned) without requiring that you reconfigure the clients.
- *Client-side partitioning.* In this model, the client application contains logic (possibly in the form of a library) that routes requests to the appropriate Redis server. This approach can be used with Azure Cache for Redis. Create multiple Azure Cache for Redis (one for each data partition) and implement the client-side logic that routes the requests to the correct cache. If the partitioning scheme changes (if additional Azure Cache for Redis are created, for example), client applications might need to be reconfigured.
- *Proxy-assisted partitioning.* In this scheme, client applications send requests to an intermediary proxy service which understands how the data is partitioned and then routes the request to the appropriate Redis server. This approach can also be used with Azure Cache for Redis; the proxy service can be implemented as an Azure cloud service. This approach requires an additional level of complexity to implement the service, and requests might take longer to perform than using client-side partitioning.

The page Partitioning: how to split data among multiple Redis instances on the Redis website provides further information about implementing partitioning with Redis.

Implement Redis cache client applications

Redis supports client applications written in numerous programming languages. If you are building new applications by using the .NET Framework, the recommended approach is to use the StackExchange.Redis client library. This library provides a .NET Framework object model that abstracts the details for connecting to a Redis server, sending commands, and receiving responses. It is available in Visual Studio as a NuGet package. You can use this same library to connect to an Azure Cache for Redis, or a custom Redis cache hosted on a VM.

To connect to a Redis server you use the static `Connect` method of the `ConnectionMultiplexer` class. The connection that this method creates is designed to be used throughout the lifetime of the client application, and the same connection can be used by multiple concurrent threads. Do not reconnect and disconnect each time you perform a Redis operation because this can degrade performance.

You can specify the connection parameters, such as the address of the Redis host and the password. If you are using Azure Cache for Redis, the password is either the primary or secondary key that is generated for Azure Cache for Redis by using the Azure portal.

After you have connected to the Redis server, you can obtain a handle on the Redis database that acts as the cache. The Redis connection provides the `GetDatabase` method to do this. You can then retrieve items from the cache and store data in the cache by using the `StringGet` and `StringSet` methods. These methods expect a key as a parameter, and return the item either in the cache that has a matching value (`StringGet`) or add the item to the cache with this key (`StringSet`).

Depending on the location of the Redis server, many operations might incur some latency while a request is

transmitted to the server and a response is returned to the client. The StackExchange library provides asynchronous versions of many of the methods that it exposes to help client applications remain responsive. These methods support the Task-based Asynchronous pattern in the .NET Framework.

The following code snippet shows a method named `RetrieveItem`. It illustrates an implementation of the cache-aside pattern based on Redis and the StackExchange library. The method takes a string key value and attempts to retrieve the corresponding item from the Redis cache by calling the `StringGetAsync` method (the asynchronous version of `StringGet`).

If the item is not found, it is fetched from the underlying data source using the `GetItemFromDataSourceAsync` method (which is a local method and not part of the StackExchange library). It's then added to the cache by using the `StringSetAsync` method so it can be retrieved more quickly next time.

```
// Connect to the Azure Redis cache
ConfigurationOptions config = new ConfigurationOptions();
config.EndPoints.Add("<your DNS name>.redis.cache.windows.net");
config.Password = "<Redis cache key from management portal>";
ConnectionMultiplexer redisHostConnection = ConnectionMultiplexer.Connect(config);
IDatabase cache = redisHostConnection.GetDatabase();
...
private async Task<string> RetrieveItem(string itemKey)
{
    // Attempt to retrieve the item from the Redis cache
    string itemValue = await cache.StringGetAsync(itemKey);

    // If the value returned is null, the item was not found in the cache
    // So retrieve the item from the data source and add it to the cache
    if (itemValue == null)
    {
        itemValue = await GetItemFromDataSourceAsync(itemKey);
        await cache.StringSetAsync(itemKey, itemValue);
    }

    // Return the item
    return itemValue;
}
```

The `StringGet` and `StringSet` methods are not restricted to retrieving or storing string values. They can take any item that is serialized as an array of bytes. If you need to save a .NET object, you can serialize it as a byte stream and use the `StringSet` method to write it to the cache.

Similarly, you can read an object from the cache by using the `StringGet` method and deserializing it as a .NET object. The following code shows a set of extension methods for the IDatabase interface (the `GetDatabase` method of a Redis connection returns an `IDatabase` object), and some sample code that uses these methods to read and write a `BlogPost` object to the cache:

```
public static class RedisCacheExtensions
{
    public static async Task<T> GetAsync<T>(this IDatabase cache, string key)
    {
        return Deserialize<T>(await cache.StringGetAsync(key));
    }

    public static async Task<object> GetAsync(this IDatabase cache, string key)
    {
        return Deserialize<object>(await cache.StringGetAsync(key));
    }

    public static async Task SetAsync(this IDatabase cache, string key, object value)
    {
        await cache.StringSetAsync(key, Serialize(value));
    }

    static byte[] Serialize(object o)
    {
        byte[] objectDataAsStream = null;

        if (o != null)
        {
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            using (MemoryStream memoryStream = new MemoryStream())
            {
                binaryFormatter.Serialize(memoryStream, o);
                objectDataAsStream = memoryStream.ToArray();
            }
        }

        return objectDataAsStream;
    }

    static T Deserialize<T>(byte[] stream)
    {
        T result = default(T);

        if (stream != null)
        {
            BinaryFormatter binaryFormatter = new BinaryFormatter();
            using (MemoryStream memoryStream = new MemoryStream(stream))
            {
                result = (T)binaryFormatter.Deserialize(memoryStream);
            }
        }

        return result;
    }
}
```

The following code illustrates a method named `RetrieveBlogPost` that uses these extension methods to read and write a serializable `BlogPost` object to the cache following the cache-aside pattern:

```
// The BlogPost type
[Serializable]
public class BlogPost
{
    private HashSet<string> tags;

    public BlogPost(int id, string title, int score, IEnumerable<string> tags)
    {
        this.Id = id;
        this.Title = title;
        this.Score = score;
        this.tags = new HashSet<string>(tags);
    }

    public int Id { get; set; }
    public string Title { get; set; }
    public int Score { get; set; }
    public ICollection<string> Tags => this.tags;
}
...
private async Task<BlogPost> RetrieveBlogPost(string blogPostKey)
{
    BlogPost blogPost = await cache.GetAsync<BlogPost>(blogPostKey);
    if (blogPost == null)
    {
        blogPost = await GetBlogPostFromDataSourceAsync(blogPostKey);
        await cache.SetAsync(blogPostKey, blogPost);
    }

    return blogPost;
}
```

Redis supports command pipelining if a client application sends multiple asynchronous requests. Redis can multiplex the requests using the same connection rather than receiving and responding to commands in a strict sequence.

This approach helps to reduce latency by making more efficient use of the network. The following code snippet shows an example that retrieves the details of two customers concurrently. The code submits two requests and then performs some other processing (not shown) before waiting to receive the results. The `Wait` method of the cache object is similar to the .NET Framework `Task.Wait` method:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
var task1 = cache.StringGetAsync("customer:1");
var task2 = cache.StringGetAsync("customer:2");
...
var customer1 = cache.Wait(task1);
var customer2 = cache.Wait(task2);
```

For additional information on writing client applications that can the Azure Cache for Redis, see the Azure Cache for Redis documentation. More information is also available at StackExchange.Redis.

The page Pipelines and multiplexers on the same website provides more information about asynchronous operations and pipelining with Redis and the StackExchange library.

Using Redis caching

The simplest use of Redis for caching concerns is key-value pairs where the value is an uninterpreted string of arbitrary length that can contain any binary data. (It is essentially an array of bytes that can be treated as a string). This scenario was illustrated in the section Implement Redis Cache client applications earlier in this

article.

Note that keys also contain uninterpreted data, so you can use any binary information as the key. The longer the key is, however, the more space it will take to store, and the longer it will take to perform lookup operations. For usability and ease of maintenance, design your keyspace carefully and use meaningful (but not verbose) keys.

For example, use structured keys such as "customer:100" to represent the key for the customer with ID 100 rather than simply "100". This scheme enables you to easily distinguish between values that store different data types. For example, you could also use the key "orders:100" to represent the key for the order with ID 100.

Apart from one-dimensional binary strings, a value in a Redis key-value pair can also hold more structured information, including lists, sets (sorted and unsorted), and hashes. Redis provides a comprehensive command set that can manipulate these types, and many of these commands are available to .NET Framework applications through a client library such as StackExchange. The page An introduction to Redis data types and abstractions on the Redis website provides a more detailed overview of these types and the commands that you can use to manipulate them.

This section summarizes some common use cases for these data types and commands.

Perform atomic and batch operations

Redis supports a series of atomic get-and-set operations on string values. These operations remove the possible race hazards that might occur when using separate `GET` and `SET` commands. The operations that are available include:

- `INCR`, `INCRBY`, `DECR`, and `DECRBY`, which perform atomic increment and decrement operations on integer numeric data values. The StackExchange library provides overloaded versions of the `IDatabase.StringIncrementAsync` and `IDatabase.StringDecrementAsync` methods to perform these operations and return the resulting value that is stored in the cache. The following code snippet illustrates how to use these methods:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
await cache.StringSetAsync("data:counter", 99);
...
long oldValue = await cache.StringIncrementAsync("data:counter");
// Increment by 1 (the default)
// oldValue should be 100

long newValue = await cache.StringDecrementAsync("data:counter", 50);
// Decrement by 50
// newValue should be 50
```

- `GETSET`, which retrieves the value that's associated with a key and changes it to a new value. The StackExchange library makes this operation available through the `IDatabase.StringGetSetAsync` method. The code snippet below shows an example of this method. This code returns the current value that's associated with the key "data:counter" from the previous example. Then it resets the value for this key back to zero, all as part of the same operation:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string oldValue = await cache.StringGetSetAsync("data:counter", 0);
```

- `MGET` and `MSET`, which can return or change a set of string values as a single operation. The `IDatabase.StringGetAsync` and `IDatabase.StringSetAsync` methods are overloaded to support this functionality, as shown in the following example:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Create a list of key-value pairs
var keysAndValues =
    new List<KeyValuePair<RedisKey, RedisValue>>()
    {
        new KeyValuePair<RedisKey, RedisValue>("data:key1", "value1"),
        new KeyValuePair<RedisKey, RedisValue>("data:key99", "value2"),
        new KeyValuePair<RedisKey, RedisValue>("data:key322", "value3")
    };

// Store the list of key-value pairs in the cache
cache.StringSet(keysAndValues.ToArray());
...
// Find all values that match a list of keys
RedisKey[] keys = { "data:key1", "data:key99", "data:key322"};
// values should contain { "value1", "value2", "value3" }
RedisValue[] values = cache.StringGet(keys);
```

You can also combine multiple operations into a single Redis transaction as described in the Redis transactions and batches section earlier in this article. The StackExchange library provides support for transactions through the `ITransaction` interface.

You create an `ITransaction` object by using the `IDatabase.CreateTransaction` method. You invoke commands to the transaction by using the methods provided by the `ITransaction` object.

The `ITransaction` interface provides access to a set of methods that's similar to those accessed by the `IDatabase` interface, except that all the methods are asynchronous. This means that they are only performed when the `ITransaction.Execute` method is invoked. The value that's returned by the `ITransaction.Execute` method indicates whether the transaction was created successfully (true) or if it failed (false).

The following code snippet shows an example that increments and decrements two counters as part of the same transaction:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
ITransaction transaction = cache.CreateTransaction();
var tx1 = transaction.StringIncrementAsync("data:counter1");
var tx2 = transaction.StringDecrementAsync("data:counter2");
bool result = transaction.Execute();
Console.WriteLine("Transaction {0}", result ? "succeeded" : "failed");
Console.WriteLine("Result of increment: {0}", tx1.Result);
Console.WriteLine("Result of decrement: {0}", tx2.Result);
```

Remember that Redis transactions are unlike transactions in relational databases. The `Execute` method simply queues all the commands that comprise the transaction to be run, and if any of them is malformed then the transaction is stopped. If all the commands have been queued successfully, each command runs asynchronously.

If any command fails, the others still continue processing. If you need to verify that a command has completed successfully, you must fetch the results of the command by using the **Result** property of the corresponding task, as shown in the example above. Reading the **Result** property will block the calling thread until the task has completed.

For more information, see Transactions in Redis.

When performing batch operations, you can use the `IBatch` interface of the StackExchange library. This

interface provides access to a set of methods similar to those accessed by the `IDatabase` interface, except that all the methods are asynchronous.

You create an `IBatch` object by using the `IDatabase.CreateBatch` method, and then run the batch by using the `IBatch.Execute` method, as shown in the following example. This code simply sets a string value, increments and decrements the same counters used in the previous example, and displays the results:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
IBatch batch = cache.CreateBatch();
batch.StringSetAsync("data:key1", 11);
var t1 = batch.StringIncrementAsync("data:counter1");
var t2 = batch.StringDecrementAsync("data:counter2");
batch.Execute();
Console.WriteLine("{0}", t1.Result);
Console.WriteLine("{0}", t2.Result);
```

It is important to understand that unlike a transaction, if a command in a batch fails because it is malformed, the other commands might still run. The `IBatch.Execute` method does not return any indication of success or failure.

Perform fire and forget cache operations

Redis supports fire and forget operations by using command flags. In this situation, the client simply initiates an operation but has no interest in the result and does not wait for the command to be completed. The example below shows how to perform the INCR command as a fire and forget operation:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
await cache.StringSetAsync("data:key1", 99);
...
cache.StringIncrement("data:key1", flags: CommandFlags.FireAndForget);
```

Specify automatically expiring keys

When you store an item in a Redis cache, you can specify a timeout after which the item will be automatically removed from the cache. You can also query how much more time a key has before it expires by using the `TTL` command. This command is available to StackExchange applications by using the `IDatabase.KeyTimeToLive` method.

The following code snippet shows how to set an expiration time of 20 seconds on a key, and query the remaining lifetime of the key:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Add a key with an expiration time of 20 seconds
await cache.StringSetAsync("data:key1", 99, TimeSpan.FromSeconds(20));
...
// Query how much time a key has left to live
// If the key has already expired, the KeyTimeToLive function returns a null
TimeSpan? expiry = cache.KeyTimeToLive("data:key1");
```

You can also set the expiration time to a specific date and time by using the EXPIRE command, which is available in the StackExchange library as the `KeyExpireAsync` method:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Add a key with an expiration date of midnight on 1st January 2015
await cache.StringSetAsync("data:key1", 99);
await cache.KeyExpireAsync("data:key1",
    new DateTime(2015, 1, 1, 0, 0, 0, DateTimeKind.Utc));
...
```

TIP

You can manually remove an item from the cache by using the DEL command, which is available through the StackExchange library as the `IDatabase.KeyDeleteAsync` method.

Use tags to cross-correlate cached items

A Redis set is a collection of multiple items that share a single key. You can create a set by using the SADD command. You can retrieve the items in a set by using the SMEMBERS command. The StackExchange library implements the SADD command with the `IDatabase.SetAddAsync` method, and the SMEMBERS command with the `IDatabase.SetMembersAsync` method.

You can also combine existing sets to create new sets by using the SDIFF (set difference), SINTER (set intersection), and SUNION (set union) commands. The StackExchange library unifies these operations in the `IDatabase.SetCombineAsync` method. The first parameter to this method specifies the set operation to perform.

The following code snippets show how sets can be useful for quickly storing and retrieving collections of related items. This code uses the `BlogPost` type that was described in the section Implement Redis Cache Client Applications earlier in this article.

A `BlogPost` object contains four fields—an ID, a title, a ranking score, and a collection of tags. The first code snippet below shows the sample data that's used for populating a C# list of `BlogPost` objects:

```
List<string[]> tags = new List<string[]>
{
    new[] { "iot","csharp" },
    new[] { "iot","azure","csharp" },
    new[] { "csharp","git","big data" },
    new[] { "iot","git","database" },
    new[] { "database","git" },
    new[] { "csharp","database" },
    new[] { "iot" },
    new[] { "iot","database","git" },
    new[] { "azure","database","big data","git","csharp" },
    new[] { "azure" }
};

List<BlogPost> posts = new List<BlogPost>();
int blogKey = 1;
int numberOfPosts = 20;
Random random = new Random();
for (int i = 0; i < numberOfPosts; i++)
{
    blogKey++;
    posts.Add(new BlogPost(
        blogKey,                  // Blog post ID
        string.Format(CultureInfo.InvariantCulture, "Blog Post #{0}",
            blogKey),             // Blog post title
        random.Next(100, 10000),  // Ranking score
        tags[i % tags.Count]));   // Tags--assigned from a collection
                                  // in the tags list
}
```

You can store the tags for each `BlogPost` object as a set in a Redis cache and associate each set with the ID of the `BlogPost`. This enables an application to quickly find all the tags that belong to a specific blog post. To enable searching in the opposite direction and find all blog posts that share a specific tag, you can create another set that holds the blog posts referencing the tag ID in the key:

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
// Tags are easily represented as Redis Sets
foreach (BlogPost post in posts)
{
    string redisKey = string.Format(CultureInfo.InvariantCulture,
        "blog:posts:{0}:tags", post.Id);
    // Add tags to the blog post in Redis
    await cache.SetAddAsync(
        redisKey, post.Tags.Select(s => (RedisValue)s).ToArray());

    // Now do the inverse so we can figure out which blog posts have a given tag
    foreach (var tag in post.Tags)
    {
        await cache.SetAddAsync(string.Format(CultureInfo.InvariantCulture,
            "tag:{0}:blog:posts", tag), post.Id);
    }
}
```

These structures enable you to perform many common queries very efficiently. For example, you can find and display all of the tags for blog post 1 like this:

```
// Show the tags for blog post #1
foreach (var value in await cache.SetMembersAsync("blog:posts:1:tags"))
{
    Console.WriteLine(value);
}
```

You can find all tags that are common to blog post 1 and blog post 2 by performing a set intersection operation, as follows:

```
// Show the tags in common for blog posts #1 and #2
foreach (var value in await cache.SetCombineAsync(SetOperation.Intersect, new RedisKey[]
    { "blog:posts:1:tags", "blog:posts:2:tags" }))
{
    Console.WriteLine(value);
}
```

And you can find all blog posts that contain a specific tag:

```
// Show the ids of the blog posts that have the tag "iot".
foreach (var value in await cache.SetMembersAsync("tag:iot:blog:posts"))
{
    Console.WriteLine(value);
}
```

Find recently accessed items

A common task required of many applications is to find the most recently accessed items. For example, a blogging site might want to display information about the most recently read blog posts.

You can implement this functionality by using a Redis list. A Redis list contains multiple items that share the same key. The list acts as a double-ended queue. You can push items to either end of the list by using the LPUSH (left push) and RPUSH (right push) commands. You can retrieve items from either end of the list by using the LPOP and RPOP commands. You can also return a set of elements by using the LRANGE and RRANGE commands.

The code snippets below show how you can perform these operations by using the StackExchange library. This code uses the `BlogPost` type from the previous examples. As a blog post is read by a user, the `IDatabase.ListLeftPushAsync` method pushes the title of the blog post onto a list that's associated with the key "blog:recent_posts" in the Redis cache.

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string redisKey = "blog:recent_posts";
BlogPost blogPost = ...; // Reference to the blog post that has just been read
await cache.ListLeftPushAsync(
    redisKey, blogPost.Title); // Push the blog post onto the list
```

As more blog posts are read, their titles are pushed onto the same list. The list is ordered by the sequence in which the titles have been added. The most recently read blog posts are toward the left end of the list. (If the same blog post is read more than once, it will have multiple entries in the list.)

You can display the titles of the most recently read posts by using the `IDatabase.ListRange` method. This method takes the key that contains the list, a starting point, and an ending point. The following code retrieves the titles of the 10 blog posts (items from 0 to 9) at the left-most end of the list:

```
// Show latest ten posts
foreach (string postTitle in await cache.ListRangeAsync(redisKey, 0, 9))
{
    Console.WriteLine(postTitle);
}
```

Note that the `ListRangeAsync` method does not remove items from the list. To do this, you can use the `IDatabase.ListLeftPopAsync` and `IDatabase.ListRightPopAsync` methods.

To prevent the list from growing indefinitely, you can periodically cull items by trimming the list. The code snippet below shows you how to remove all but the five left-most items from the list:

```
await cache.ListTrimAsync(redisKey, 0, 5);
```

Implement a leader board

By default, the items in a set are not held in any specific order. You can create an ordered set by using the ZADD command (the `IDatabase.SortedSetAdd` method in the StackExchange library). The items are ordered by using a numeric value called a score, which is provided as a parameter to the command.

The following code snippet adds the title of a blog post to an ordered list. In this example, each blog post also has a score field that contains the ranking of the blog post.

```
ConnectionMultiplexer redisHostConnection = ...;
IDatabase cache = redisHostConnection.GetDatabase();
...
string redisKey = "blog:post_rankings";
BlogPost blogPost = ...; // Reference to a blog post that has just been rated
await cache.SortedSetAddAsync(redisKey, blogPost.Title, blogPost.Score);
```

You can retrieve the blog post titles and scores in ascending score order by using the `IDatabase.SortedSetRangeByRankWithScores` method:

```
foreach (var post in await cache.SortedSetRangeByRankWithScoresAsync(redisKey))
{
    Console.WriteLine(post);
}
```

> **NOTE**
> The StackExchange library also provides the `IDatabase.SortedSetRangeByRankAsync` method, which returns the data in score order, but does not return the scores.

You can also retrieve items in descending order of scores, and limit the number of items that are returned by providing additional parameters to the `IDatabase.SortedSetRangeByRankWithScoresAsync` method. The next example displays the titles and scores of the top 10 ranked blog posts:

```
foreach (var post in await cache.SortedSetRangeByRankWithScoresAsync(
                              redisKey, 0, 9, Order.Descending))
{
    Console.WriteLine(post);
}
```

The next example uses the `IDatabase.SortedSetRangeByScoreWithScoresAsync` method, which you can use to limit

the items that are returned to those that fall within a given score range:

```
// Blog posts with scores between 5000 and 100000
foreach (var post in await cache.SortedSetRangeByScoreWithScoresAsync(
                              redisKey, 5000, 100000))
{
    Console.WriteLine(post);
}
```

Message by using channels

Apart from acting as a data cache, a Redis server provides messaging through a high-performance publisher/subscriber mechanism. Client applications can subscribe to a channel, and other applications or services can publish messages to the channel. Subscribing applications will then receive these messages and can process them.

Redis provides the SUBSCRIBE command for client applications to use to subscribe to channels. This command expects the name of one or more channels on which the application will accept messages. The StackExchange library includes the `ISubscription` interface, which enables a .NET Framework application to subscribe and publish to channels.

You create an `ISubscription` object by using the `GetSubscriber` method of the connection to the Redis server. Then you listen for messages on a channel by using the `SubscribeAsync` method of this object. The following code example shows how to subscribe to a channel named "messages:blogPosts":

```
ConnectionMultiplexer redisHostConnection = ...;
ISubscriber subscriber = redisHostConnection.GetSubscriber();
...
await subscriber.SubscribeAsync("messages:blogPosts", (channel, message) => Console.WriteLine("Title is:
{0}", message));
```

The first parameter to the `Subscribe` method is the name of the channel. This name follows the same conventions that are used by keys in the cache. The name can contain any binary data, although it is advisable to use relatively short, meaningful strings to help ensure good performance and maintainability.

Note also that the namespace used by channels is separate from that used by keys. This means you can have channels and keys that have the same name, although this may make your application code more difficult to maintain.

The second parameter is an Action delegate. This delegate runs asynchronously whenever a new message appears on the channel. This example simply displays the message on the console (the message will contain the title of a blog post).

To publish to a channel, an application can use the Redis PUBLISH command. The StackExchange library provides the `IServer.PublishAsync` method to perform this operation. The next code snippet shows how to publish a message to the "messages:blogPosts" channel:

```
ConnectionMultiplexer redisHostConnection = ...;
ISubscriber subscriber = redisHostConnection.GetSubscriber();
...
BlogPost blogPost = ...;
subscriber.PublishAsync("messages:blogPosts", blogPost.Title);
```

There are several points you should understand about the publish/subscribe mechanism:

- Multiple subscribers can subscribe to the same channel, and they will all receive the messages that are published to that channel.

- Subscribers only receive messages that have been published after they have subscribed. Channels are not buffered, and once a message is published, the Redis infrastructure pushes the message to each subscriber and then removes it.
- By default, messages are received by subscribers in the order in which they are sent. In a highly active system with a large number of messages and many subscribers and publishers, guaranteed sequential delivery of messages can slow performance of the system. If each message is independent and the order is unimportant, you can enable concurrent processing by the Redis system, which can help to improve responsiveness. You can achieve this in a StackExchange client by setting the PreserveAsyncOrder of the connection used by the subscriber to false:

```
ConnectionMultiplexer redisHostConnection = ...;
redisHostConnection.PreserveAsyncOrder = false;
ISubscriber subscriber = redisHostConnection.GetSubscriber();
```

Serialization considerations

When you choose a serialization format, consider tradeoffs between performance, interoperability, versioning, compatibility with existing systems, data compression, and memory overhead. When you are evaluating performance, remember that benchmarks are highly dependent on context. They may not reflect your actual workload, and may not consider newer libraries or versions. There is no single "fastest" serializer for all scenarios.

Some options to consider include:

- Protocol Buffers (also called protobuf) is a serialization format developed by Google for serializing structured data efficiently. It uses strongly typed definition files to define message structures. These definition files are then compiled to language-specific code for serializing and deserializing messages. Protobuf can be used over existing RPC mechanisms, or it can generate an RPC service.
- Apache Thrift uses a similar approach, with strongly typed definition files and a compilation step to generate the serialization code and RPC services.
- Apache Avro provides similar functionality to Protocol Buffers and Thrift, but there is no compilation step. Instead, serialized data always includes a schema that describes the structure.
- JSON is an open standard that uses human-readable text fields. It has broad cross-platform support. JSON does not use message schemas. Being a text-based format, it is not very efficient over the wire. In some cases, however, you may be returning cached items directly to a client via HTTP, in which case storing JSON could save the cost of deserializing from another format and then serializing to JSON.
- BSON is a binary serialization format that uses a structure similar to JSON. BSON was designed to be lightweight, easy to scan, and fast to serialize and deserialize, relative to JSON. Payloads are comparable in size to JSON. Depending on the data, a BSON payload may be smaller or larger than a JSON payload. BSON has some additional data types that are not available in JSON, notably BinData (for byte arrays) and Date.
- MessagePack is a binary serialization format that is designed to be compact for transmission over the wire. There are no message schemas or message type checking.
- Bond is a cross-platform framework for working with schematized data. It supports cross-language serialization and deserialization. Notable differences from other systems listed here are support for inheritance, type aliases, and generics.
- gRPC is an open-source RPC system developed by Google. By default, it uses Protocol Buffers as its definition language and underlying message interchange format.

Related patterns and guidance

The following patterns might also be relevant to your scenario when you implement caching in your applications:

- Cache-aside pattern: This pattern describes how to load data on demand into a cache from a data store. This pattern also helps to maintain consistency between data that's held in the cache and the data in the original data store.
- The Sharding pattern provides information about implementing horizontal partitioning to help improve scalability when storing and accessing large volumes of data.

More information

- Azure Cache for Redis documentation
- Azure Cache for Redis FAQ
- Task-based Asynchronous pattern
- Redis documentation
- StackExchange.Redis
- Data partitioning guide

Best practices for using content delivery networks (CDNs)

11/19/2021 • 8 minutes to read • Edit Online

A content delivery network (CDN) is a distributed network of servers that can efficiently deliver web content to users. CDNs store cached content on edge servers that are close to end users to minimize latency.

CDNs are typically used to deliver static content such as images, style sheets, documents, client-side scripts, and HTML pages. The major advantages of using a CDN are lower latency and faster delivery of content to users, regardless of their geographical location in relation to the datacenter where the application is hosted. CDNs can also help to reduce load on a web application, because the application does not have to service requests for the content that is hosted in the CDN.

In Azure, the Azure Content Delivery Network is a global CDN solution for delivering high-bandwidth content that is hosted in Azure or any other location. Using Azure CDN, you can cache publicly available objects loaded from Azure blob storage, a web application, virtual machine, any publicly accessible web server.

This topic describes some general best practices and considerations when using a CDN. For more information, see Azure CDN.

How and why a CDN is used

Typical uses for a CDN include:

- Delivering static resources for client applications, often from a website. These resources can be images, style sheets, documents, files, client-side scripts, HTML pages, HTML fragments, or any other content that the server does not need to modify for each request. The application can create items at runtime and make them available to the CDN (for example, by creating a list of current news headlines), but it does not do so for each request.
- Delivering public static and shared content to devices such as mobile phones and tablet computers. The application itself is a web service that offers an API to clients running on the various devices. The CDN can also deliver static datasets (via the web service) for the clients to use, perhaps to generate the client UI. For example, the CDN could be used to distribute JSON or XML documents.
- Serving entire websites that consist of only public static content to clients, without requiring any dedicated compute resources.
- Streaming video files to the client on demand. Video benefits from the low latency and reliable

connectivity available from the globally located datacenters that offer CDN connections. Microsoft Azure Media Services (AMS) integrates with Azure CDN to deliver content directly to the CDN for further distribution. For more information, see Streaming endpoints overview.

- Generally improving the experience for users, especially those located far from the datacenter hosting the application. These users might otherwise suffer higher latency. A large proportion of the total size of the content in a web application is often static, and using the CDN can help to maintain performance and overall user experience while eliminating the requirement to deploy the application to multiple datacenters. For a list of Azure CDN node locations, see Azure CDN POP Locations.
- Supporting IoT (Internet of Things) solutions. The huge numbers of devices and appliances involved in an IoT solution could easily overwhelm an application if it had to distribute firmware updates directly to each device.
- Coping with peaks and surges in demand without requiring the application to scale, avoiding the consequent increase in running costs. For example, when an update to an operating system is released for a hardware device such as a specific model of router, or for a consumer device such as a smart TV, there will be a huge peak in demand as it is downloaded by millions of users and devices over a short period.

Challenges

There are several challenges to take into account when planning to use a CDN.

- **Deployment**. Decide the origin from which the CDN fetches the content, and whether you need to deploy the content in more than one storage system. Take into account the process for deploying static content and resources. For example, you may need to implement a separate step to load content into Azure blob storage.
- **Versioning and cache-control**. Consider how you will update static content and deploy new versions. Understand how the CDN performs caching and time-to-live (TTL). For Azure CDN, see How caching works.
- **Testing**. It can be difficult to perform local testing of your CDN settings when developing and testing an application locally or in a staging environment.
- **Search engine optimization (SEO)**. Content such as images and documents are served from a different domain when you use the CDN. This can have an effect on SEO for this content.
- **Content security**. Not all CDNs offer any form of access control for the content. Some CDN services, including Azure CDN, support token-based authentication to protect CDN content. For more information, see Securing Azure Content Delivery Network assets with token authentication.
- **Client security**. Clients might connect from an environment that does not allow access to resources on the CDN. This could be a security-constrained environment that limits access to only a set of known sources, or one that prevents loading of resources from anything other than the page origin. A fallback implementation is required to handle these cases.
- **Resilience**. The CDN is a potential single point of failure for an application.

Scenarios where a CDN may be less useful include:

- If the content has a low hit rate, it might be accessed only few times while it is valid (determined by its time-to-live setting).
- If the data is private, such as for large enterprises or supply chain ecosystems.

General guidelines and good practices

Using a CDN is a good way to minimize the load on your application, and maximize availability and performance. Consider adopting this strategy for all of the appropriate content and resources your application uses. Consider the points in the following sections when designing your strategy to use a CDN.

Deployment

Static content may need to be provisioned and deployed independently from the application if you do not include it in the application deployment package or process. Consider how this will affect the versioning approach you use to manage both the application components and the static resource content.

Consider using bundling and minification techniques to reduce load times for clients. Bundling combines multiple files into a single file. Minification removes unnecessary characters from scripts and CSS files without altering functionality.

If you need to deploy the content to an additional location, this will be an extra step in the deployment process. If the application updates the content for the CDN, perhaps at regular intervals or in response to an event, it must store the updated content in any additional locations as well as the endpoint for the CDN.

Consider how you will handle local development and testing when some static content is expected to be served from a CDN. For example, you could predeploy the content to the CDN as part of your build script. Alternatively, use compile directives or flags to control how the application loads the resources. For example, in debug mode, the application could load static resources from a local folder. In release mode, the application would use the CDN.

Consider the options for file compression, such as gzip (GNU zip). Compression may be performed on the origin server by the web application hosting or directly on the edge servers by the CDN. For more information, see Improve performance by compressing files in Azure CDN.

Routing and versioning

You may need to use different CDN instances at various times. For example, when you deploy a new version of the application you may want to use a new CDN and retain the old CDN (holding content in an older format) for previous versions. If you use Azure blob storage as the content origin, you can create a separate storage account or a separate container and point the CDN endpoint to it.

Do not use the query string to denote different versions of the application in links to resources on the CDN because, when retrieving content from Azure blob storage, the query string is part of the resource name (the blob name). This approach can also affect how the client caches resources.

Deploying new versions of static content when you update an application can be a challenge if the previous resources are cached on the CDN. For more information, see the section on cache control, below.

Consider restricting the CDN content access by country/region. Azure CDN allows you to filter requests based on the country or region of origin and restrict the content delivered. For more information, see Restrict access to your content by country/region.

Cache control

Consider how to manage caching within the system. For example, in Azure CDN, you can set global caching rules, and then set custom caching for particular origin endpoints. You can also control how caching is performed in a CDN by sending cache-directive headers at the origin.

For more information, see How caching works.

To prevent objects from being available on the CDN, you can delete them from the origin, remove or delete the CDN endpoint, or in the case of blob storage, make the container or blob private. However, items are not removed from the CDN until the time-to-live expires. You can also manually purge a CDN endpoint.

Security

The CDN can deliver content over HTTPS (SSL), by using the certificate provided by the CDN, as well as over standard HTTP. To avoid browser warnings about mixed content, you might need to use HTTPS to request static content that is displayed in pages loaded through HTTPS.

If you deliver static assets such as font files by using the CDN, you might encounter same-origin policy issues if you use an *XMLHttpRequest* call to request these resources from a different domain. Many web browsers prevent cross-origin resource sharing (CORS) unless the web server is configured to set the appropriate response headers. You can configure the CDN to support CORS by using one of the following methods:

- Configure the CDN to add CORS headers to the responses. For more information, see Using Azure CDN with CORS.
- If the origin is Azure blob storage, add CORS rules to the storage endpoint. For more information, see Cross-Origin Resource Sharing (CORS) Support for the Azure Storage Services.
- Configure the application to set the CORS headers. For example, see Enabling Cross-Origin Requests (CORS) in the ASP.NET Core documentation.

CDN fallback

Consider how your application will cope with a failure or temporary unavailability of the CDN. Client applications may be able to use copies of the resources that were cached locally (on the client) during previous requests, or you can include code that detects failure and instead requests resources from the origin (the application folder or Azure blob container that holds the resources) if the CDN is unavailable.

Horizontal, vertical, and functional data partitioning

11/19/2021 • 17 minutes to read • Edit Online

In many large-scale solutions, data is divided into *partitions* that can be managed and accessed separately. Partitioning can improve scalability, reduce contention, and optimize performance. It can also provide a mechanism for dividing data by usage pattern. For example, you can archive older data in cheaper data storage.

However, the partitioning strategy must be chosen carefully to maximize the benefits while minimizing adverse effects.

> **NOTE**
>
> In this article, the term *partitioning* means the process of physically dividing data into separate data stores. It is not the same as SQL Server table partitioning.

Why partition data?

- **Improve scalability**. When you scale up a single database system, it will eventually reach a physical hardware limit. If you divide data across multiple partitions, each hosted on a separate server, you can scale out the system almost indefinitely.
- **Improve performance**. Data access operations on each partition take place over a smaller volume of data. Correctly done, partitioning can make your system more efficient. Operations that affect more than one partition can run in parallel.
- **Improve security**. In some cases, you can separate sensitive and nonsensitive data into different partitions and apply different security controls to the sensitive data.
- **Provide operational flexibility**. Partitioning offers many opportunities for fine-tuning operations, maximizing administrative efficiency, and minimizing cost. For example, you can define different strategies for management, monitoring, backup and restore, and other administrative tasks based on the importance of the data in each partition.
- **Match the data store to the pattern of use**. Partitioning allows each partition to be deployed on a different type of data store, based on cost and the built-in features that data store offers. For example, large binary data can be stored in blob storage, while more structured data can be held in a document database. See Choose the right data store.
- **Improve availability**. Separating data across multiple servers avoids a single point of failure. If one instance fails, only the data in that partition is unavailable. Operations on other partitions can continue. For managed PaaS data stores, this consideration is less relevant, because these services are designed with built-in redundancy.

Designing partitions

There are three typical strategies for partitioning data:

- **Horizontal partitioning** (often called *sharding*). In this strategy, each partition is a separate data store, but all partitions have the same schema. Each partition is known as a *shard* and holds a specific subset of the data, such as all the orders for a specific set of customers.
- **Vertical partitioning**. In this strategy, each partition holds a subset of the fields for items in the data

store. The fields are divided according to their pattern of use. For example, frequently accessed fields might be placed in one vertical partition and less frequently accessed fields in another.

- **Functional partitioning**. In this strategy, data is aggregated according to how it is used by each bounded context in the system. For example, an e-commerce system might store invoice data in one partition and product inventory data in another.

These strategies can be combined, and we recommend that you consider them all when you design a partitioning scheme. For example, you might divide data into shards and then use vertical partitioning to further subdivide the data in each shard.

Horizontal partitioning (sharding)

Figure 1 shows horizontal partitioning or sharding. In this example, product inventory data is divided into shards based on the product key. Each shard holds the data for a contiguous range of shard keys (A-G and H-Z), organized alphabetically. Sharding spreads the load over more computers, which reduces contention and improves performance.

Figure 1 - Horizontally partitioning (sharding) data based on a partition key.

The most important factor is the choice of a sharding key. It can be difficult to change the key after the system is in operation. The key must ensure that data is partitioned to spread the workload as evenly as possible across the shards.

The shards don't have to be the same size. It's more important to balance the number of requests. Some shards might be very large, but each item has a low number of access operations. Other shards might be smaller, but each item is accessed much more frequently. It's also important to ensure that a single shard does not exceed the scale limits (in terms of capacity and processing resources) of the data store.

Avoid creating "hot" partitions that can affect performance and availability. For example, using the first letter of a customer's name causes an unbalanced distribution, because some letters are more common. Instead, use a hash of a customer identifier to distribute data more evenly across partitions.

Choose a sharding key that minimizes any future requirements to split large shards, coalesce small shards into larger partitions, or change the schema. These operations can be very time consuming, and might require taking one or more shards offline while they are performed.

If shards are replicated, it might be possible to keep some of the replicas online while others are split, merged, or reconfigured. However, the system might need to limit the operations that can be performed during the reconfiguration. For example, the data in the replicas might be marked as read-only to prevent data inconsistences.

For more information about horizontal partitioning, see sharding pattern.

Vertical partitioning

The most common use for vertical partitioning is to reduce the I/O and performance costs associated with

fetching items that are frequently accessed. Figure 2 shows an example of vertical partitioning. In this example, different properties of an item are stored in different partitions. One partition holds data that is accessed more frequently, including product name, description, and price. Another partition holds inventory data: the stock count and last-ordered date.

Figure 2 - Vertically partitioning data by its pattern of use.

In this example, the application regularly queries the product name, description, and price when displaying the product details to customers. Stock count and last- ordered date are held in a separate partition because these two items are commonly used together.

Other advantages of vertical partitioning:

- Relatively slow-moving data (product name, description, and price) can be separated from the more dynamic data (stock level and last ordered date). Slow moving data is a good candidate for an application to cache in memory.
- Sensitive data can be stored in a separate partition with additional security controls.
- Vertical partitioning can reduce the amount of concurrent access that's needed.

Vertical partitioning operates at the entity level within a data store, partially normalizing an entity to break it down from a *wide* item to a set of *narrow* items. It is ideally suited for column-oriented data stores such as HBase and Cassandra. If the data in a collection of columns is unlikely to change, you can also consider using column stores in SQL Server.

Functional partitioning

When it's possible to identify a bounded context for each distinct business area in an application, functional partitioning is a way to improve isolation and data access performance. Another common use for functional partitioning is to separate read-write data from read-only data. Figure 3 shows an overview of functional partitioning where inventory data is separated from customer data.

Figure 3 - Functionally partitioning data by bounded context or subdomain.

This partitioning strategy can help reduce data access contention across different parts of a system.

Designing partitions for scalability

It's vital to consider size and workload for each partition and balance them so that data is distributed to achieve maximum scalability. However, you must also partition the data so that it does not exceed the scaling limits of a single partition store.

Follow these steps when designing partitions for scalability:

1. Analyze the application to understand the data access patterns, such as the size of the result set returned by each query, the frequency of access, the inherent latency, and the server-side compute processing requirements. In many cases, a few major entities will demand most of the processing resources.
2. Use this analysis to determine the current and future scalability targets, such as data size and workload. Then distribute the data across the partitions to meet the scalability target. For horizontal partitioning, choosing the right shard key is important to make sure distribution is even. For more information, see the sharding pattern.
3. Make sure each partition has enough resources to handle the scalability requirements, in terms of data size and throughput. Depending on the data store, there might be a limit on the amount of storage space, processing power, or network bandwidth per partition. If the requirements are likely to exceed these limits, you may need to refine your partitioning strategy or split data out further, possibly combining two or more strategies.
4. Monitor the system to verify that data is distributed as expected and that the partitions can handle the load. Actual usage does not always match what an analysis predicts. If so, it might be possible to rebalance the partitions, or else redesign some parts of the system to gain the required balance.

Some cloud environments allocate resources in terms of infrastructure boundaries. Ensure that the limits of your selected boundary provide enough room for any anticipated growth in the volume of data, in terms of data storage, processing power, and bandwidth.

For example, if you use Azure table storage, there is a limit to the volume of requests that can be handled by a single partition in a particular period of time. (For more information, see Azure storage scalability and performance targets.) A busy shard might require more resources than a single partition can handle. If so, the shard might need to be repartitioned to spread the load. If the total size or throughput of these tables exceeds the capacity of a storage account, you might need to create additional storage accounts and spread the tables

across these accounts.

Designing partitions for query performance

Query performance can often be boosted by using smaller data sets and by running parallel queries. Each partition should contain a small proportion of the entire data set. This reduction in volume can improve the performance of queries. However, partitioning is not an alternative for designing and configuring a database appropriately. For example, make sure that you have the necessary indexes in place.

Follow these steps when designing partitions for query performance:

1. Examine the application requirements and performance:
   - Use business requirements to determine the critical queries that must always perform quickly.
   - Monitor the system to identify any queries that perform slowly.
   - Find which queries are performed most frequently. Even if a single query has a minimal cost, the cumulative resource consumption could be significant.
2. Partition the data that is causing slow performance:
   - Limit the size of each partition so that the query response time is within target.
   - If you use horizontal partitioning, design the shard key so that the application can easily select the right partition. This prevents the query from having to scan through every partition.
   - Consider the location of a partition. If possible, try to keep data in partitions that are geographically close to the applications and users that access it.
3. If an entity has throughput and query performance requirements, use functional partitioning based on that entity. If this still doesn't satisfy the requirements, apply horizontal partitioning as well. In most cases, a single partitioning strategy will suffice, but in some cases it is more efficient to combine both strategies.
4. Consider running queries in parallel across partitions to improve performance.

Designing partitions for availability

Partitioning data can improve the availability of applications by ensuring that the entire dataset does not constitute a single point of failure and that individual subsets of the dataset can be managed independently.

Consider the following factors that affect availability:

How critical the data is to business operations. Identify which data is critical business information, such as transactions, and which data is less critical operational data, such as log files.

- Consider storing critical data in highly available partitions with an appropriate backup plan.
- Establish separate management and monitoring procedures for the different datasets.
- Place data that has the same level of criticality in the same partition so that it can be backed up together at an appropriate frequency. For example, partitions that hold transaction data might need to be backed up more frequently than partitions that hold logging or trace information.

How individual partitions can be managed. Designing partitions to support independent management and maintenance provides several advantages. For example:

- If a partition fails, it can be recovered independently without applications that access data in other partitions.
- Partitioning data by geographical area allows scheduled maintenance tasks to occur at off-peak hours for each location. Ensure that partitions are not too large to prevent any planned maintenance from being completed during this period.

Whether to replicate critical data across partitions. This strategy can improve availability and performance, but can also introduce consistency issues. It takes time to synchronize changes with every replica. During this period, different partitions will contain different data values.

Application design considerations

Partitioning adds complexity to the design and development of your system. Consider partitioning as a fundamental part of system design even if the system initially only contains a single partition. If you address partitioning as an afterthought, it will be more challenging because you already have a live system to maintain:

- Data access logic will need to be modified.
- Large quantities of existing data may need to be migrated, to distribute it across partitions.
- Users expect to be able to continue using the system during the migration.

In some cases, partitioning is not considered important because the initial dataset is small and can be easily handled by a single server. This might be true for some workloads, but many commercial systems need to expand as the number of users increases.

Moreover, it's not only large data stores that benefit from partitioning. For example, a small data store might be heavily accessed by hundreds of concurrent clients. Partitioning the data in this situation can help to reduce contention and improve throughput.

Consider the following points when you design a data partitioning scheme:

Minimize cross-partition data access operations. Where possible, keep data for the most common database operations together in each partition to minimize cross-partition data access operations. Querying across partitions can be more time-consuming than querying within a single partition, but optimizing partitions for one set of queries might adversely affect other sets of queries. If you must query across partitions, minimize query time by running parallel queries and aggregating the results within the application. (This approach might not be possible in some cases, such as when the result from one query is used in the next query.)

Consider replicating static reference data. If queries use relatively static reference data, such as postal code tables or product lists, consider replicating this data in all of the partitions to reduce separate lookup operations in different partitions. This approach can also reduce the likelihood of the reference data becoming a "hot" dataset, with heavy traffic from across the entire system. However, there is an additional cost associated with synchronizing any changes to the reference data.

Minimize cross-partition joins. Where possible, minimize requirements for referential integrity across vertical and functional partitions. In these schemes, the application is responsible for maintaining referential integrity across partitions. Queries that join data across multiple partitions are inefficient because the application typically needs to perform consecutive queries based on a key and then a foreign key. Instead, consider replicating or de-normalizing the relevant data. If cross-partition joins are necessary, run parallel queries over the partitions and join the data within the application.

Embrace eventual consistency. Evaluate whether strong consistency is actually a requirement. A common approach in distributed systems is to implement eventual consistency. The data in each partition is updated separately, and the application logic ensures that the updates are all completed successfully. It also handles the inconsistencies that can arise from querying data while an eventually consistent operation is running.

Consider how queries locate the correct partition. If a query must scan all partitions to locate the required data, there is a significant impact on performance, even when multiple parallel queries are running. With vertical and functional partitioning, queries can naturally specify the partition. Horizontal partitioning, on the other hand, can make locating an item difficult, because every shard has the same schema. A typical solution to maintain a map that is used to look up the shard location for specific items. This map can be implemented in the sharding logic of the application, or maintained by the data store if it supports transparent sharding.

Consider periodically rebalancing shards. With horizontal partitioning, rebalancing shards can help distribute the data evenly by size and by workload to minimize hotspots, maximize query performance, and work around physical storage limitations. However, this is a complex task that often requires the use of a custom tool or process.

Replicate partitions. If you replicate each partition, it provides additional protection against failure. If a single replica fails, queries can be directed toward a working copy.

If you reach the physical limits of a partitioning strategy, you might need to extend the scalability to a different level. For example, if partitioning is at the database level, you might need to locate or replicate partitions in multiple databases. If partitioning is already at the database level, and physical limitations are an issue, it might mean that you need to locate or replicate partitions in multiple hosting accounts.

Avoid transactions that access data in multiple partitions. Some data stores implement transactional consistency and integrity for operations that modify data, but only when the data is located in a single partition. If you need transactional support across multiple partitions, you will probably need to implement this as part of your application logic because most partitioning systems do not provide native support.

All data stores require some operational management and monitoring activity. The tasks can range from loading data, backing up and restoring data, reorganizing data, and ensuring that the system is performing correctly and efficiently.

Consider the following factors that affect operational management:

- **How to implement appropriate management and operational tasks when the data is partitioned**. These tasks might include backup and restore, archiving data, monitoring the system, and other administrative tasks. For example, maintaining logical consistency during backup and restore operations can be a challenge.
- **How to load the data into multiple partitions and add new data that's arriving from other sources**. Some tools and utilities might not support sharded data operations such as loading data into the correct partition.
- **How to archive and delete the data on a regular basis**. To prevent the excessive growth of partitions, you need to archive and delete data on a regular basis (such as monthly). It might be necessary to transform the data to match a different archive schema.
- **How to locate data integrity issues**. Consider running a periodic process to locate any data integrity issues, such as data in one partition that references missing information in another. The process can either attempt to fix these issues automatically or generate a report for manual review.

Rebalancing partitions

As a system matures, you might have to adjust the partitioning scheme. For example, individual partitions might start getting a disproportionate volume of traffic and become hot, leading to excessive contention. Or you might have underestimated the volume of data in some partitions, causing some partitions to approach capacity limits.

Some data stores, such as Cosmos DB, can automatically rebalance partitions. In other cases, rebalancing is an administrative task that consists of two stages:

1. Determine a new partitioning strategy.
    - Which partitions need to be split (or possibly combined)?
    - What is the new partition key?
2. Migrate data from the old partitioning scheme to the new set of partitions.

Depending on the data store, you might be able to migrate data between partitions while they are in use. This is

called *online migration*. If that's not possible, you might need to make partitions unavailable while the data is relocated (*offline migration*).

Offline migration

Offline migration is typically simpler because it reduces the chances of contention occurring. Conceptually, offline migration works as follows:

1. Mark the partition offline.
2. Split-merge and move the data to the new partitions.
3. Verify the data.
4. Bring the new partitions online.
5. Remove the old partition.

Optionally, you can mark a partition as read-only in step 1, so that applications can still read the data while it is being moved.

Online migration

Online migration is more complex to perform but less disruptive. The process is similar to offline migration, except the original partition is not marked offline. Depending on the granularity of the migration process (for example, item by item versus shard by shard), the data access code in the client applications might have to handle reading and writing data that's held in two locations, the original partition and the new partition.

Related patterns

The following design patterns might be relevant to your scenario:

- The sharding pattern describes some common strategies for sharding data.
- The index table pattern shows how to create secondary indexes over data. An application can quickly retrieve data with this approach, by using queries that do not reference the primary key of a collection.
- The materialized view pattern describes how to generate prepopulated views that summarize data to support fast query operations. This approach can be useful in a partitioned data store if the partitions that contain the data being summarized are distributed across multiple sites.

Next steps

- Learn about partitioning strategies for specific Azure services. See Data partitioning strategies

Data partitioning strategies

11/19/2021 • 31 minutes to read • Edit Online

This article describes some strategies for partitioning data in various Azure data stores. For general guidance about when to partition data and best practices, see Data partitioning.

Partitioning Azure SQL Database

A single SQL database has a limit to the volume of data that it can contain. Throughput is constrained by architectural factors and the number of concurrent connections that it supports.

Elastic pools support horizontal scaling for a SQL database. Using elastic pools, you can partition your data into shards that are spread across multiple SQL databases. You can also add or remove shards as the volume of data that you need to handle grows and shrinks. Elastic pools can also help reduce contention by distributing the load across databases.

Each shard is implemented as a SQL database. A shard can hold more than one dataset (called a *shardlet*). Each database maintains metadata that describes the shardlets that it contains. A shardlet can be a single data item, or a group of items that share the same shardlet key. For example, in a multitenant application, the shardlet key can be the tenant ID, and all data for a tenant can be held in the same shardlet.

Client applications are responsible for associating a dataset with a shardlet key. A separate SQL database acts as a global shard map manager. This database has a list of all the shards and shardlets in the system. The application connects to the shard map manager database to obtain a copy of the shard map. It caches the shard map locally, and uses the map to route data requests to the appropriate shard. This functionality is hidden behind a series of APIs that are contained in the Elastic Database client library, which is available for Java and .NET.

For more information about elastic pools, see Scaling out with Azure SQL Database.

To reduce latency and improve availability, you can replicate the global shard map manager database. With the Premium pricing tiers, you can configure active geo-replication to continuously copy data to databases in different regions.

Alternatively, use Azure SQL Data Sync or Azure Data Factory to replicate the shard map manager database across regions. This form of replication runs periodically and is more suitable if the shard map changes infrequently, and does not require Premium tier.

Elastic Database provides two schemes for mapping data to shardlets and storing them in shards:

- A **list shard map** associates a single key to a shardlet. For example, in a multitenant system, the data for each tenant can be associated with a unique key and stored in its own shardlet. To guarantee isolation, each shardlet can be held within its own shard.

- A **range shard map** associates a set of contiguous key values to a shardlet. For example, you can group the data for a set of tenants (each with their own key) within the same shardlet. This scheme is less expensive than the first, because tenants share data storage, but has less isolation.

A single shard can contain the data for several shardlets. For example, you can use list shardlets to store data for different non-contiguous tenants in the same shard. You can also mix range shardlets and list shardlets in the same shard, although they will be addressed through different maps. The following diagram shows this approach:

Elastic pools make it possible to add and remove shards as the volume of data shrinks and grows. Client applications can create and delete shards dynamically, and transparently update the shard map manager. However, removing a shard is a destructive operation that also requires deleting all the data in that shard.

If an application needs to split a shard into two separate shards or combine shards, use the split-merge tool. This tool runs as an Azure web service, and migrates data safely between shards.

The partitioning scheme can significantly affect the performance of your system. It can also affect the rate at which shards have to be added or removed, or that data must be repartitioned across shards. Consider the following points:

- Group data that is used together in the same shard, and avoid operations that access data from multiple shards. A shard is a SQL database in its own right, and cross-database joins must be performed on the client side.

  Although SQL Database does not support cross-database joins, you can use the Elastic Database tools to perform multi-shard queries. A multi-shard query sends individual queries to each database and merges the results.

- Don't design a system that has dependencies between shards. Referential integrity constraints, triggers, and stored procedures in one database cannot reference objects in another.

- If you have reference data that is frequently used by queries, consider replicating this data across shards. This approach can remove the need to join data across databases. Ideally, such data should be static or slow-moving, to minimize the replication effort and reduce the chances of it becoming stale.
- Shardlets that belong to the same shard map should have the same schema. This rule is not enforced by SQL Database, but data management and querying becomes very complex if each shardlet has a different schema. Instead, create separate shard maps for each schema. Remember that data belonging to different shardlets can be stored in the same shard.
- Transactional operations are only supported for data within a shard, and not across shards. Transactions can span shardlets as long as they are part of the same shard. Therefore, if your business logic needs to perform transactions, either store the data in the same shard or implement eventual consistency.
- Place shards close to the users that access the data in those shards. This strategy helps reduce latency.
- Avoid having a mixture of highly active and relatively inactive shards. Try to spread the load evenly across shards. This might require hashing the sharding keys. If you are geo-locating shards, make sure that the hashed keys map to shardlets held in shards stored close to the users that access that data.

Partitioning Azure table storage

Azure table storage is a key-value store that's designed around partitioning. All entities are stored in a partition, and partitions are managed internally by Azure table storage. Each entity stored in a table must provide a two-part key that includes:

- **The partition key**. This is a string value that determines the partition where Azure table storage will place the entity. All entities with the same partition key are stored in the same partition.
- **The row key**. This is a string value that identifies the entity within the partition. All entities within a partition are sorted lexically, in ascending order, by this key. The partition key/row key combination must be unique for each entity and cannot exceed 1 KB in length.

If an entity is added to a table with a previously unused partition key, Azure table storage creates a new partition for this entity. Other entities with the same partition key will be stored in the same partition.

This mechanism effectively implements an automatic scale-out strategy. Each partition is stored on the same server in an Azure datacenter to help ensure that queries that retrieve data from a single partition run quickly.

Microsoft has published scalability targets for Azure Storage. If your system is likely to exceed these limits, consider splitting entities into multiple tables. Use vertical partitioning to divide the fields into the groups that are most likely to be accessed together.

The following diagram shows the logical structure of an example storage account. The storage account contains three tables: Customer Info, Product Info, and Order Info.

Each table has multiple partitions.

- In the Customer Info table, the data is partitioned according to the city where the customer is located. The row key contains the customer ID.
- In the Product Info table, products are partitioned by product category, and the row key contains the product number.
- In the Order Info table, the orders are partitioned by order date, and the row key specifies the time the order was received. All data is ordered by the row key in each partition.

Consider the following points when you design your entities for Azure table storage:

- Select a partition key and row key by how the data is accessed. Choose a partition key/row key combination that supports the majority of your queries. The most efficient queries retrieve data by specifying the partition key and the row key. Queries that specify a partition key and a range of row keys can be completed by scanning a single partition. This is relatively fast because the data is held in row key order. If queries don't specify which partition to scan, every partition must be scanned.

- If an entity has one natural key, then use it as the partition key and specify an empty string as the row key. If an entity has a composite key consisting of two properties, select the slowest changing property as the partition key and the other as the row key. If an entity has more than two key properties, use a concatenation of properties to provide the partition and row keys.

- If you regularly perform queries that look up data by using fields other than the partition and row keys, consider implementing the Index Table pattern, or consider using a different data store that supports indexing, such as Cosmos DB.

- If you generate partition keys by using a monotonic sequence (such as "0001", "0002", "0003") and each partition only contains a limited amount of data, Azure table storage can physically group these partitions together on the same server. Azure Storage assumes that the application is most likely to perform queries across a contiguous range of partitions (range queries) and is optimized for this case.

However, this approach can lead to hotspots, because all insertions of new entities are likely to be concentrated at one end the contiguous range. It can also reduce scalability. To spread the load more evenly, consider hashing the partition key.

- Azure table storage supports transactional operations for entities that belong to the same partition. An application can perform multiple insert, update, delete, replace, or merge operations as an atomic unit, as long as the transaction doesn't include more than 100 entities and the payload of the request doesn't exceed 4 MB. Operations that span multiple partitions are not transactional, and might require you to implement eventual consistency. For more information about table storage and transactions, see Performing entity group transactions.
- Consider the granularity of the partition key:
  - Using the same partition key for every entity results in a single partition that's held on one server. This prevents the partition from scaling out and focuses the load on a single server. As a result, this approach is only suitable for storing a small number of entities. However, it does ensure that all entities can participate in entity group transactions.
  - Using a unique partition key for every entity causes the table storage service to create a separate partition for each entity, possibly resulting in a large number of small partitions. This approach is more scalable than using a single partition key, but entity group transactions are not possible. Also, queries that fetch more than one entity might involve reading from more than one server. However, if the application performs range queries, then using a monotonic sequence for the partition keys might help to optimize these queries.
  - Sharing the partition key across a subset of entities makes it possible to group related entities in the same partition. Operations that involve related entities can be performed by using entity group transactions, and queries that fetch a set of related entities can be satisfied by accessing a single server.

For more information, see Azure storage table design guide and Scalable partitioning strategy.

Partitioning Azure Blob Storage

Azure Blob Storage makes it possible to hold large binary objects. Use block blobs in scenarios when you need to upload or download large volumes of data quickly. Use page blobs for applications that require random rather than serial access to parts of the data.

Each blob (either block or page) is held in a container in an Azure storage account. You can use containers to group related blobs that have the same security requirements. This grouping is logical rather than physical. Inside a container, each blob has a unique name.

The partition key for a blob is account name + container name + blob name. The partition key is used to partition data into ranges and these ranges are load-balanced across the system. Blobs can be distributed across many servers in order to scale out access to them, but a single blob can only be served by a single server.

If your naming scheme uses timestamps or numerical identifiers, it can lead to excessive traffic going to one partition, limiting the system from effectively load balancing. For instance, if you have daily operations that use a blob object with a timestamp such as *yyyy-mm-dd*, all the traffic for that operation would go to a single partition server. Instead, consider prefixing the name with a three-digit hash. For more information, see Partition Naming Convention.

The actions of writing a single block or page are atomic, but operations that span blocks, pages, or blobs are not. If you need to ensure consistency when performing write operations across blocks, pages, and blobs, take out a write lock by using a blob lease.

Partitioning Azure storage queues

Azure storage queues enable you to implement asynchronous messaging between processes. An Azure storage account can contain any number of queues, and each queue can contain any number of messages. The only limitation is the space that's available in the storage account. The maximum size of an individual message is 64 KB. If you require messages bigger than this, then consider using Azure Service Bus queues instead.

Each storage queue has a unique name within the storage account that contains it. Azure partitions queues based on the name. All messages for the same queue are stored in the same partition, which is controlled by a single server. Different queues can be managed by different servers to help balance the load. The allocation of queues to servers is transparent to applications and users.

In a large-scale application, don't use the same storage queue for all instances of the application because this approach might cause the server that's hosting the queue to become a hot spot. Instead, use different queues for different functional areas of the application. Azure storage queues do not support transactions, so directing messages to different queues should have little effect on messaging consistency.

An Azure storage queue can handle up to 2,000 messages per second. If you need to process messages at a greater rate than this, consider creating multiple queues. For example, in a global application, create separate storage queues in separate storage accounts to handle application instances that are running in each region.

Partitioning Azure Service Bus

Azure Service Bus uses a message broker to handle messages that are sent to a Service Bus queue or topic. By default, all messages that are sent to a queue or topic are handled by the same message broker process. This architecture can place a limitation on the overall throughput of the message queue. However, you can also partition a queue or topic when it is created. You do this by setting the *EnablePartitioning* property of the queue or topic description to *true*.

A partitioned queue or topic is divided into multiple fragments, each of which is backed by a separate message store and message broker. Service Bus takes responsibility for creating and managing these fragments. When an application posts a message to a partitioned queue or topic, Service Bus assigns the message to a fragment for that queue or topic. When an application receives a message from a queue or subscription, Service Bus checks each fragment for the next available message and then passes it to the application for processing.

This structure helps distribute the load across message brokers and message stores, increasing scalability and improving availability. If the message broker or message store for one fragment is temporarily unavailable, Service Bus can retrieve messages from one of the remaining available fragments.

Service Bus assigns a message to a fragment as follows:

- If the message belongs to a session, all messages with the same value for the *SessionId* property are sent to the same fragment.
- If the message does not belong to a session, but the sender has specified a value for the *PartitionKey* property, then all messages with the same *PartitionKey* value are sent to the same fragment.

  > **NOTE**
  >
  > If the *SessionId* and *PartitionKey* properties are both specified, then they must be set to the same value or the message will be rejected.

- If the *SessionId* and *PartitionKey* properties for a message are not specified, but duplicate detection is enabled, the *MessageId* property will be used. All messages with the same *MessageId* will be directed to the same fragment.

- If messages do not include a *SessionId, PartitionKey,* or *MessageId* property, then Service Bus assigns messages to fragments sequentially. If a fragment is unavailable, Service Bus will move on to the next. This means that a temporary fault in the messaging infrastructure does not cause the message-send operation to fail.

Consider the following points when deciding if or how to partition a Service Bus message queue or topic:

- Service Bus queues and topics are created within the scope of a Service Bus namespace. Service Bus currently allows up to 100 partitioned queues or topics per namespace.
- Each Service Bus namespace imposes quotas on the available resources, such as the number of subscriptions per topic, the number of concurrent send and receive requests per second, and the maximum number of concurrent connections that can be established. These quotas are documented in Service Bus quotas. If you expect to exceed these values, then create additional namespaces with their own queues and topics, and spread the work across these namespaces. For example, in a global application, create separate namespaces in each region and configure application instances to use the queues and topics in the nearest namespace.
- Messages that are sent as part of a transaction must specify a partition key. This can be a *SessionId*, *PartitionKey*, or *MessageId* property. All messages that are sent as part of the same transaction must specify the same partition key because they must be handled by the same message broker process. You cannot send messages to different queues or topics within the same transaction.
- Partitioned queues and topics can't be configured to be automatically deleted when they become idle.
- Partitioned queues and topics can't currently be used with the Advanced Message Queuing Protocol (AMQP) if you are building cross-platform or hybrid solutions.

Partitioning Cosmos DB

Azure Cosmos DB is a NoSQL database that can store JSON documents using the Azure Cosmos DB SQL API. A document in a Cosmos DB database is a JSON-serialized representation of an object or other piece of data. No fixed schemas are enforced except that every document must contain a unique ID.

Documents are organized into collections. You can group related documents together in a collection. For example, in a system that maintains blog postings, you can store the contents of each blog post as a document in a collection. You can also create collections for each subject type. Alternatively, in a multitenant application, such as a system where different authors control and manage their own blog posts, you can partition blogs by author and create separate collections for each author. The storage space that's allocated to collections is elastic and can shrink or grow as needed.

Cosmos DB supports automatic partitioning of data based on an application-defined partition key. A *logical partition* is a partition that stores all the data for a single partition key value. All documents that share the same value for the partition key are placed within the same logical partition. Cosmos DB distributes values according to hash of the partition key. A logical partition has a maximum size of 10 GB. Therefore, the choice of the partition key is an important decision at design time. Choose a property with a wide range of values and even access patterns. For more information, see Partition and scale in Azure Cosmos DB.

> **NOTE**
>
> Each Cosmos DB database has a *performance level* that determines the amount of resources it gets. A performance level is associated with a *request unit* (RU) rate limit. The RU rate limit specifies the volume of resources that's reserved and available for exclusive use by that collection. The cost of a collection depends on the performance level that's selected for that collection. The higher the performance level (and RU rate limit) the higher the charge. You can adjust the performance level of a collection by using the Azure portal. For more information, see Request Units in Azure Cosmos DB.

If the partitioning mechanism that Cosmos DB provides is not sufficient, you may need to shard the data at the application level. Document collections provide a natural mechanism for partitioning data within a single database. The simplest way to implement sharding is to create a collection for each shard. Containers are logical resources and can span one or more servers. Fixed-size containers have a maximum limit of 10 GB and 10,000 RU/s throughput. Unlimited containers do not have a maximum storage size, but must specify a partition key. With application sharding, the client application must direct requests to the appropriate shard, usually by implementing its own mapping mechanism based on some attributes of the data that define the shard key.

All databases are created in the context of a Cosmos DB database account. A single account can contain several databases, and it specifies in which regions the databases are created. Each account also enforces its own access control. You can use Cosmos DB accounts to geo-locate shards (collections within databases) close to the users who need to access them, and enforce restrictions so that only those users can connect to them.

Consider the following points when deciding how to partition data with the Cosmos DB SQL API:

- **The resources available to a Cosmos DB database are subject to the quota limitations of the account**. Each database can hold a number of collections, and each collection is associated with a performance level that governs the RU rate limit (reserved throughput) for that collection. For more information, see Azure subscription and service limits, quotas, and constraints.
- **Each document must have an attribute that can be used to uniquely identify that document within the collection in which it is held**. This attribute is different from the shard key, which defines which collection holds the document. A collection can contain a large number of documents. In theory, it's limited only by the maximum length of the document ID. The document ID can be up to 255 characters.
- **All operations against a document are performed within the context of a transaction. Transactions are scoped to the collection in which the document is contained.** If an operation fails, the work that it has performed is rolled back. While a document is subject to an operation, any changes that are made are subject to snapshot-level isolation. This mechanism guarantees that if, for example, a request to create a new document fails, another user who's querying the database simultaneously will not see a partial document that is then removed.
- **Database queries are also scoped to the collection level**. A single query can retrieve data from only one collection. If you need to retrieve data from multiple collections, you must query each collection individually and merge the results in your application code.
- **Cosmos DB supports programmable items that can all be stored in a collection alongside documents**. These include stored procedures, user-defined functions, and triggers (written in JavaScript). These items can access any document within the same collection. Furthermore, these items run either inside the scope of the ambient transaction (in the case of a trigger that fires as the result of a create, delete, or replace operation performed against a document), or by starting a new transaction (in the case of a stored procedure that is run as the result of an explicit client request). If the code in a programmable item throws an exception, the transaction is rolled back. You can use stored procedures and triggers to maintain integrity and consistency between documents, but these documents must all be part of the same collection.
- **The collections that you intend to hold in the databases should be unlikely to exceed the throughput limits defined by the performance levels of the collections**. For more information, see Request Units in Azure Cosmos DB. If you anticipate reaching these limits, consider splitting collections across databases in different accounts to reduce the load per collection.

Partitioning Azure Search

The ability to search for data is often the primary method of navigation and exploration that's provided by many web applications. It helps users find resources quickly (for example, products in an e-commerce application) based on combinations of search criteria. The Azure Search service provides full-text search capabilities over

web content, and includes features such as type-ahead, suggested queries based on near matches, and faceted navigation. For more information, see What is Azure Search?.

Azure Search stores searchable content as JSON documents in a database. You define indexes that specify the searchable fields in these documents and provide these definitions to Azure Search. When a user submits a search request, Azure Search uses the appropriate indexes to find matching items.

To reduce contention, the storage that's used by Azure Search can be divided into 1, 2, 3, 4, 6, or 12 partitions, and each partition can be replicated up to 6 times. The product of the number of partitions multiplied by the number of replicas is called the *search unit* (SU). A single instance of Azure Search can contain a maximum of 36 SUs (a database with 12 partitions only supports a maximum of 3 replicas).

You are billed for each SU that is allocated to your service. As the volume of searchable content increases or the rate of search requests grows, you can add SUs to an existing instance of Azure Search to handle the extra load. Azure Search itself distributes the documents evenly across the partitions. No manual partitioning strategies are currently supported.

Each partition can contain a maximum of 15 million documents or occupy 300 GB of storage space (whichever is smaller). You can create up to 50 indexes. The performance of the service varies and depends on the complexity of the documents, the available indexes, and the effects of network latency. On average, a single replica (1 SU) should be able to handle 15 queries per second (QPS), although we recommend performing benchmarking with your own data to obtain a more precise measure of throughput. For more information, see Service limits in Azure Search.

> **NOTE**
> You can store a limited set of data types in searchable documents, including strings, Booleans, numeric data, datetime data, and some geographical data. For more information, see the page Supported data types (Azure Search) on the Microsoft website.

You have limited control over how Azure Search partitions data for each instance of the service. However, in a global environment you might be able to improve performance and reduce latency and contention further by partitioning the service itself using either of the following strategies:

- Create an instance of Azure Search in each geographic region, and ensure that client applications are directed toward the nearest available instance. This strategy requires that any updates to searchable content are replicated in a timely manner across all instances of the service.
- Create two tiers of Azure Search:
  - A local service in each region that contains the data that's most frequently accessed by users in that region. Users can direct requests here for fast but limited results.
  - A global service that encompasses all the data. Users can direct requests here for slower but more complete results.

This approach is most suitable when there is a significant regional variation in the data that's being searched.

Partitioning Azure Cache for Redis

Azure Cache for Redis provides a shared caching service in the cloud that's based on the Redis key-value data store. As its name implies, Azure Cache for Redis is intended as a caching solution. Use it only for holding transient data and not as a permanent data store. Applications that use Azure Cache for Redis should be able to continue functioning if the cache is unavailable. Azure Cache for Redis supports primary/secondary replication to provide high availability, but currently limits the maximum cache size to 53 GB. If you need more space than this, you must create additional caches. For more information, see Azure Cache for Redis.

Partitioning a Redis data store involves splitting the data across instances of the Redis service. Each instance constitutes a single partition. Azure Cache for Redis abstracts the Redis services behind a façade and does not expose them directly. The simplest way to implement partitioning is to create multiple Azure Cache for Redis instances and spread the data across them.

You can associate each data item with an identifier (a partition key) that specifies which cache stores the data item. The client application logic can then use this identifier to route requests to the appropriate partition. This scheme is very simple, but if the partitioning scheme changes (for example, if additional Azure Cache for Redis instances are created), client applications might need to be reconfigured.

Native Redis (not Azure Cache for Redis) supports server-side partitioning based on Redis clustering. In this approach, you can divide the data evenly across servers by using a hashing mechanism. Each Redis server stores metadata that describes the range of hash keys that the partition holds, and also contains information about which hash keys are located in the partitions on other servers.

Client applications simply send requests to any of the participating Redis servers (probably the closest one). The Redis server examines the client request. If it can be resolved locally, it performs the requested operation. Otherwise it forwards the request on to the appropriate server.

This model is implemented by using Redis clustering, and is described in more detail on the Redis cluster tutorial page on the Redis website. Redis clustering is transparent to client applications. Additional Redis servers can be added to the cluster (and the data can be repartitioned) without requiring that you reconfigure the clients.

> **IMPORTANT**
>
> Azure Cache for Redis currently supports Redis clustering in premium tier only.

The page Partitioning: how to split data among multiple Redis instances on the Redis website provides more information about implementing partitioning with Redis. The remainder of this section assumes that you are implementing client-side or proxy-assisted partitioning.

Consider the following points when deciding how to partition data with Azure Cache for Redis:

- Azure Cache for Redis is not intended to act as a permanent data store, so whatever partitioning scheme you implement, your application code must be able to retrieve data from a location that's not the cache.
- Data that is frequently accessed together should be kept in the same partition. Redis is a powerful key-value store that provides several highly optimized mechanisms for structuring data. These mechanisms can be one of the following:
  - Simple strings (binary data up to 512 MB in length)
  - Aggregate types such as lists (which can act as queues and stacks)
  - Sets (ordered and unordered)
  - Hashes (which can group related fields together, such as the items that represent the fields in an object)
- The aggregate types enable you to associate many related values with the same key. A Redis key identifies a list, set, or hash rather than the data items that it contains. These types are all available with Azure Cache for Redis and are described by the Data types page on the Redis website. For example, in part of an e-commerce system that tracks the orders that are placed by customers, the details of each customer can be stored in a Redis hash that is keyed by using the customer ID. Each hash can hold a collection of order IDs for the customer. A separate Redis set can hold the orders, again structured as hashes, and keyed by using the order ID. Figure 8 shows this structure. Note that Redis does not implement any form of referential integrity, so it is the developer's responsibility to maintain the relationships between customers and orders.

Figure 8. Suggested structure in Redis storage for recording customer orders and their details.

> **NOTE**
>
> In Redis, all keys are binary data values (like Redis strings) and can contain up to 512 MB of data. In theory, a key can contain almost any information. However, we recommend adopting a consistent naming convention for keys that is descriptive of the type of data and that identifies the entity, but is not excessively long. A common approach is to use keys of the form "entity_type:ID". For example, you can use "customer:99" to indicate the key for a customer with the ID 99.

- You can implement vertical partitioning by storing related information in different aggregations in the same database. For example, in an e-commerce application, you can store commonly accessed information about products in one Redis hash and less frequently used detailed information in another. Both hashes can use the same product ID as part of the key. For example, you can use "product: *nn*" (where *nn* is the product ID) for the product information and "product_details: *nn*" for the detailed data. This strategy can help reduce the volume of data that most queries are likely to retrieve.

- You can repartition a Redis data store, but keep in mind that it's a complex and time-consuming task. Redis clustering can repartition data automatically, but this capability is not available with Azure Cache for Redis. Therefore, when you design your partitioning scheme, try to leave sufficient free space in each partition to allow for expected data growth over time. However, remember that Azure Cache for Redis is intended to cache data temporarily, and that data held in the cache can have a limited lifetime specified as a time-to-live (TTL) value. For relatively volatile data, the TTL can be short, but for static data the TTL can be a lot longer. Avoid storing large amounts of long-lived data in the cache if the volume of this data is likely to fill the cache. You can specify an eviction policy that causes Azure Cache for Redis to remove data if space is at a premium.

  > **NOTE**
  >
  > When you use Azure Cache for Redis, you specify the maximum size of the cache (from 250 MB to 53 GB) by selecting the appropriate pricing tier. However, after an Azure Cache for Redis has been created, you cannot increase (or decrease) its size.

- Redis batches and transactions cannot span multiple connections, so all data that is affected by a batch or transaction should be held in the same database (shard).

> **NOTE**
>
> A sequence of operations in a Redis transaction is not necessarily atomic. The commands that compose a transaction are verified and queued before they run. If an error occurs during this phase, the entire queue is discarded. However, after the transaction has been successfully submitted, the queued commands run in sequence. If any command fails, only that command stops running. All previous and subsequent commands in the queue are performed. For more information, go to the Transactions page on the Redis website.

- Redis supports a limited number of atomic operations. The only operations of this type that support multiple keys and values are MGET and MSET operations. MGET operations return a collection of values for a specified list of keys, and MSET operations store a collection of values for a specified list of keys. If you need to use these operations, the key-value pairs that are referenced by the MSET and MGET commands must be stored within the same database.

Partitioning Azure Service Fabric

Azure Service Fabric is a microservices platform that provides a runtime for distributed applications in the cloud. Service Fabric supports .NET guest executables, stateful and stateless services, and containers. Stateful services provide a reliable collection to persistently store data in a key-value collection within the Service Fabric cluster. For more information about strategies for partitioning keys in a reliable collection, see guidelines and recommendations for reliable collections in Azure Service Fabric.

More information

- Overview of Azure Service Fabric is an introduction to Azure Service Fabric.
- Partition Service Fabric reliable services provides more information about reliable services in Azure Service Fabric.

Partitioning Azure Event Hubs

Azure Event Hubs is designed for data streaming at massive scale, and partitioning is built into the service to enable horizontal scaling. Each consumer only reads a specific partition of the message stream.

The event publisher is only aware of its partition key, not the partition to which the events are published. This decoupling of key and partition insulates the sender from needing to know too much about the downstream processing. (It's also possible send events directly to a given partition, but generally that's not recommended.)

Consider long-term scale when you select the partition count. After an event hub is created, you can't change the number of partitions.

For more information about using partitions in Event Hubs, see What is Event Hubs?.

For considerations about trade-offs between availability and consistency, see Availability and consistency in Event Hubs.

Message encoding considerations

11/19/2021 • 11 minutes to read • Edit Online

Many cloud applications use asynchronous messages to exchange information between components of the system. An important aspect of messaging is the format used to encode the payload data. After you choose a messaging technology, the next step is to define how the messages will be encoded. There are many options available, but the right choice depends on your use case.

This article describes some of the considerations.

Message exchange needs

A message exchange between a producer and a consumer needs:

- A shape or structure that defines the payload of the message.
- An encoding format to represent the payload.
- Serialization libraries to read and write the encoded payload.

The producer of the message defines the message shape based on the business logic and the information it wants to send to the consumer(s). To structure the shape, divide the information into discrete or related subjects (fields). Decide the characteristics of the values for those fields. Consider: What is the most efficient datatype? Will the payload always have certain fields? Will the payload have a single record or a repeated set of values?

Then, choose an encoding format depending on your need. Certain factors include the ability to create highly structured data if you need it, time taken to encode and transfer the message, and the ability to parse the payload. Depending on the encoding format, choose a serialization library that is well supported.

A consumer of the message must be aware of those decisions so that it knows how to read incoming messages.

To transfer messages, the producer serializes the message to an encoding format. At the receiving end, the consumer deserializes the payload to use the data. This way both entities share the model and as long as the shape doesn't change, messaging continues without issues. When the contract changes, the encoding format should be capable of handling the change without breaking the consumer.

Some encoding formats such as JSON are self-describing, meaning they can be parsed without referencing a schema. However, such formats tend to yield larger messages. With other formats, the data may not be parsed as easily but the messages are compact. This article highlights some factors that can help you choose a format.

Encoding format considerations

The encoding format defines how a set of structured data is represented as bytes. The type of message can influence the format choice. Messages related to business transactions most likely will contain highly structured data. Also, you may want to retrieve it later for auditing purposes. For a stream of events, you might want to read a sequence of records as quickly as possible and store it for statistical analysis.

Here are some points to consider when choosing an encoding format.

Human readability

Message encoding can be broadly divided into text-based and binary formats.

With text-based encoding, the message payload is in plain text and therefore can be inspected by a person without using any code libraries. Human readable formats are suitable for archival data. Also, because a human can read the payload, text-based formats are easier to debug and send to logs for troubleshooting errors.

The downside is that the payload tends to be larger. A common text-based format is JSON.

Encoding size

Message size impacts network I/O performance across the wire. Binary formats are more compact than text-based formats. Binary formats require serialization/deserialization libraries. The payload can't be read unless it's decoded.

Use a binary format if you want to reduce wire footprint and transfer messages faster. This category of format is recommended in scenarios where storage or network bandwidth is a concern. Options for binary formats include Apache Avro, Google Protocol Buffers (protobuf), MessagePack, and Concise Binary Object Representation (CBOR). The pros and cons of those formats are described in this section.

The disadvantage is that the payload isn't human readable. Most binary formats use complex systems that can be costly to maintain. Also, they need specialized libraries to decode, which may not be supported if you want to retrieve archival data.

Understanding the payload

A message payload arrives as a sequence of bytes. To parse this sequence, the consumer must have access to metadata that describes the data fields in the payload. There are two main approaches for storing and distributing metadata:

Tagged metadata. In some encodings, notably JSON, fields are tagged with the data type and identifier, within the body of the message. These formats are *self-describing* because they can be parsed into a dictionary of values without referring to a schema. One way for the consumer to understand the fields is to query for expected values. For example, the producer sends a payload in JSON. The consumer parses the JSON into a dictionary and checks the existence of fields to understand the payload. Another way is for the consumer to apply a data model shared by the producer. For example, if you are using a statically typed language, many JSON serialization libraries can parse a JSON string into a typed class.

Schema. A schema formally defines the structure and data fields of a message. In this model, the producer and consumer have a contract through a well-defined schema. The schema can define the data types, required/optional fields, version information, and the structure of the payload. The producer sends the payload as per the writer schema. The consumer receives the payload by applying a reader schema. The message is serialized/deserialized by using the encoding-specific libraries. There are two ways to distribute schemas:

- Store the schema as a preamble or header in the message but separate from the payload.
- Store the schema externally.

Some encoding formats define the schema and use tools that generate classes from the schema. The producer and consumer use those classes and libraries to serialize and deserialize the payload. The libraries also provide compatibility checks between the writer and reader schema. Both protobuf and Apache Avro follow that approach. The key difference is that protobuf has a language-agnostic schema definition but Avro uses compact JSON. Another difference is in the way both formats provide compatibility checks between reader and writer schemas.

Another way to store the schema externally in a schema registry. The message contains a reference to the schema and the payload. The producer sends the schema identifier in the message and the consumer retrieves the schema by specifying that identifier from an external store. Both parties use format-specific library to read and write messages. Apart from storing the schema a registry can provide compatibility checks to make sure the contract between the producer and consumer isn't broken as the schema evolves.

Before choosing an approach, decide what is more important: the transfer data size or the ability to parse the archived data later.

Storing the schema along with the payload yields a larger encoding size and is preferred for intermittent messages. Choose this approach if transferring smaller chunks of bytes is crucial or you expect a sequence of

records. The cost of maintaining an external schema store can be high.

However, if on-demand decoding of the payload is more important than size, including the schema with the payload or the tagged metadata approach guarantees decoding afterwards. There might be a significant increase in message size and may impact the cost of storage.

Schema versioning

As business requirements change, the shape is expected to change, and the schema will evolve. Versioning allows the producer to indicate schema updates that might include new features. There are two aspects to versioning:

- The consumer should be aware of the changes.

  One way is for the consumer to check all fields to determine whether the schema has changed. Another way is for the producer to publish a schema version number with the message. When the schema evolves, the producer increments the version.

- Changes must not affect or break the business logic of consumers.

  Suppose a field is added to an existing schema. If consumers using the new version get a payload as per the old version, their logic might break if they are not able to overlook the lack of the new field. Considering the reverse case, suppose a field is removed in the new schema. Consumers using the old schema might not be able to read the data.

  Encoding formats such as Avro offer the ability to define default values. In the preceding example, if the field is added with a default value, the missing field will be populated with the default value. Other formats such as protobuf provide similar functionality through required and optional fields.

Payload structure

Consider the way data is arranged in the payload. Is it a sequence of records or a discrete single payload? The payload structure can be categorized into one of these models:

- Array/dictionary/value: Defines entries that hold values in one or multi-dimensional arrays. Entries have unique key-value pairs. It can be extended to represent the complex structures. Some examples include, JSON, Apache Avro, and MessagePack.

  This layout is suitable if messages are individual encoded with different schemas. If you have multiple records, the payload can get overly redundant causing the payload to bloat.

- Tabular data: Information is divided into rows and columns. Each column indicates a field, or the subject of the information and each row contains values for those fields. This layout is efficient for a repeating set of information, such as time series data.

  CSV is one of the simplest text-based formats. It presents data as a sequence of records with a common header. For binary encoding, Apache Avro has a preamble is similar to a CSV header but generate compact encoding size.

Library support

Consider using well-known formats over a proprietary model.

Well-known formats are supported through libraries that are universally supported by the community. With specialized formats, you need specific libraries. Your business logic might have to work around some of the API design choices provided by the libraries.

For schema-based format, choose an encoding library that makes compatibility checks between the reader and writer schema. Certain encoding libraries, such as Apache Avro, expect the consumer to specify both writer and the reader schema before deserializing the message. This check ensures that the consumer is aware of the schema versions.

Interoperability

Your choice of formats might depend on the particular workload or technology ecosystem.

For example:

- Azure Stream Analytics has native support for JSON, CSV, and Avro. When using Stream Analytics, it makes sense to choose one of these formats if possible. If not, you can provide a custom deserializer, but this adds some additional complexity to your solution.
- JSON is a standard interchange format for HTTP REST APIs. If your application receives JSON payloads from clients and then places these onto a message queue for asynchronous processing, it might make sense to use JSON for the messaging, rather than re-encode into a different format.

These are just two examples of interoperability considerations. In general, standardized formats will be more interoperable than custom formats. In text-based options, JSON is one of the most interoperable.

Choices for encoding formats

Here are some popular encoding formats. Factor in the considerations before you choose a format.

JSON

JSON is an open standard (IETF RFC8259). It's a text-based format that follows the array/dictionary/value model.

JSON can be used for tagging metadata and you can parse the payload without a schema. JSON supports the option to specify optional fields, which helps with forward and backward compatibility.

The biggest advantage is that its universally available. It's most interoperable and the default encoding format for many messaging services.

Being a text-based format, it isn't efficient over the wire and not an ideal choice in cases where storage is a concern. If you're returning cached items directly to a client via HTTP, storing JSON could save the cost of deserializing from another format and then serializing to JSON.

Use JSON for single-record messages or for a sequence of messages in which each message has a different schema. Avoid using JSON for a sequence of records, such as for time-series data.

There are other variations of JSON such as BSON, which is a binary encoding aligned to work with MongoDB.

Comma-Separated Values (CSV)

CSV is a text-based tabular format. The header of the table indicates the fields. It's a preferred choice where the message contains a set of records.

The disadvantage is lack of standardization. There are many ways of expressing separators, headers, and empty fields.

Protocol Buffers (protobuf)

Google Protocol Buffers (or protobuf) is a serialization format that uses strongly typed definition files to define schemas in key/value pairs. These definition files are then compiled to language-specific classes that are used for serializing and deserializing messages.

The message contains a compressed binary small payload, which results is faster transfer. The downside is the payload isn't human readable. Also, because the schema is external, it's not recommended for cases where you have to retrieve archived data.

Apache Avro

Apache Avro is a binary serialization format that uses definition file similar to protobuf but there isn't a compilation step. Instead, serialized data always includes a schema preamble.

The preamble can hold the header or a schema identifier. Because of the smaller encoding size, Avro is recommended for streaming data. Also, because it has a header that applies to a set of records, it's a good choice for tabular data.

MessagePack

MessagePack is a binary serialization format that is designed to be compact for transmission over the wire. There are no message schemas or message type checking. This format isn't recommended for bulk storage.

CBOR

Concise Binary Object Representation (CBOR) (Specification) is a binary format that offers small encoding size. The advantage of CBOR over MessagePack is that its compliant with IETF in RFC7049.

Next steps

- Understand messaging design patterns for cloud applications.

Best practices for monitoring cloud applications

11/19/2021 • 68 minutes to read • Edit Online

Distributed applications and services running in the cloud are, by their nature, complex pieces of software that comprise many moving parts. In a production environment, it's important to be able to track the way in which users use your system, trace resource utilization, and generally monitor the health and performance of your system. You can use this information as a diagnostic aid to detect and correct issues, and also to help spot potential problems and prevent them from occurring.

Monitoring and diagnostics scenarios

You can use monitoring to gain an insight into how well a system is functioning. Monitoring is a crucial part of maintaining quality-of-service targets. Common scenarios for collecting monitoring data include:

- Ensuring that the system remains healthy.
- Tracking the availability of the system and its component elements.
- Maintaining performance to ensure that the throughput of the system does not degrade unexpectedly as the volume of work increases.
- Guaranteeing that the system meets any service-level agreements (SLAs) established with customers.
- Protecting the privacy and security of the system, users, and their data.
- Tracking the operations that are performed for auditing or regulatory purposes.
- Monitoring the day-to-day usage of the system and spotting trends that might lead to problems if they're not addressed.
- Tracking issues that occur, from initial report through to analysis of possible causes, rectification, consequent software updates, and deployment.
- Tracing operations and debugging software releases.

> **NOTE**
> This list is not intended to be comprehensive. This document focuses on these scenarios as the most common situations for performing monitoring. There might be others that are less common or are specific to your environment.

The following sections describe these scenarios in more detail. The information for each scenario is discussed in the following format:

1. A brief overview of the scenario.
2. The typical requirements of this scenario.
3. The raw instrumentation data that's required to support the scenario, and possible sources of this information.
4. How this raw data can be analyzed and combined to generate meaningful diagnostic information.

Health monitoring

A system is healthy if it is running and capable of processing requests. The purpose of health monitoring is to generate a snapshot of the current health of the system so that you can verify that all components of the system are functioning as expected.

Requirements for health monitoring

An operator should be alerted quickly (within a matter of seconds) if any part of the system is deemed to be unhealthy. The operator should be able to ascertain which parts of the system are functioning normally, and which parts are experiencing problems. System health can be highlighted through a traffic-light system:

- Red for unhealthy (the system has stopped)
- Yellow for partially healthy (the system is running with reduced functionality)
- Green for completely healthy

A comprehensive health-monitoring system enables an operator to drill down through the system to view the health status of subsystems and components. For example, if the overall system is depicted as partially healthy, the operator should be able to zoom in and determine which functionality is currently unavailable.

Data sources, instrumentation, and data-collection requirements

The raw data that's required to support health monitoring can be generated as a result of:

- Tracing execution of user requests. This information can be used to determine which requests have succeeded, which have failed, and how long each request takes.
- Synthetic user monitoring. This process simulates the steps performed by a user and follows a predefined series of steps. The results of each step should be captured.
- Logging exceptions, faults, and warnings. This information can be captured as a result of trace statements embedded into the application code, as well as retrieving information from the event logs of any services that the system references.
- Monitoring the health of any third-party services that the system uses. This monitoring might require retrieving and parsing health data that these services supply. This information might take a variety of formats.
- Endpoint monitoring. This mechanism is described in more detail in the "Availability monitoring" section.
- Collecting ambient performance information, such as background CPU utilization or I/O (including network) activity.

Analyzing health data

The primary focus of health monitoring is to quickly indicate whether the system is running. Hot analysis of the immediate data can trigger an alert if a critical component is detected as unhealthy. (It fails to respond to a consecutive series of pings, for example.) The operator can then take the appropriate corrective action.

A more advanced system might include a predictive element that performs a cold analysis over recent and current workloads. A cold analysis can spot trends and determine whether the system is likely to remain healthy or whether the system will need additional resources. This predictive element should be based on critical performance metrics, such as:

- The rate of requests directed at each service or subsystem.
- The response times of these requests.
- The volume of data flowing into and out of each service.

If the value of any metric exceeds a defined threshold, the system can raise an alert to enable an operator or autoscaling (if available) to take the preventative actions necessary to maintain system health. These actions might involve adding resources, restarting one or more services that are failing, or applying throttling to lower-priority requests.

Availability monitoring

A truly healthy system requires that the components and subsystems that compose the system are available. Availability monitoring is closely related to health monitoring. But whereas health monitoring provides an immediate view of the current health of the system, availability monitoring is concerned with tracking the availability of the system and its components to generate statistics about the uptime of the system.

In many systems, some components (such as a database) are configured with built-in redundancy to permit rapid failover in the event of a serious fault or loss of connectivity. Ideally, users should not be aware that such a failure has occurred. But from an availability monitoring perspective, it's necessary to gather as much information as possible about such failures to determine the cause and take corrective actions to prevent them from recurring.

The data that's required to track availability might depend on a number of lower-level factors. Many of these factors might be specific to the application, system, and environment. An effective monitoring system captures the availability data that corresponds to these low-level factors and then aggregates them to give an overall picture of the system. For example, in an e-commerce system, the business functionality that enables a customer to place orders might depend on the repository where order details are stored and the payment system that handles the monetary transactions for paying for these orders. The availability of the order-placement part of the system is therefore a function of the availability of the repository and the payment subsystem.

Requirements for availability monitoring

An operator should also be able to view the historical availability of each system and subsystem, and use this information to spot any trends that might cause one or more subsystems to periodically fail. (Do services start to fail at a particular time of day that corresponds to peak processing hours?)

A monitoring solution should provide an immediate and historical view of the availability or unavailability of each subsystem. It should also be capable of quickly alerting an operator when one or more services fail or when users can't connect to services. This is a matter of not only monitoring each service, but also examining the actions that each user performs if these actions fail when they attempt to communicate with a service. To some extent, a degree of connectivity failure is normal and might be due to transient errors. But it might be useful to allow the system to raise an alert for the number of connectivity failures to a specified subsystem that occur during a specific period.

Data sources, instrumentation, and data-collection requirements

As with health monitoring, the raw data that's required to support availability monitoring can be generated as a result of synthetic user monitoring and logging any exceptions, faults, and warnings that might occur. In addition, availability data can be obtained from performing endpoint monitoring. The application can expose one or more health endpoints, each testing access to a functional area within the system. The monitoring system can ping each endpoint by following a defined schedule and collect the results (success or fail).

All timeouts, network connectivity failures, and connection retry attempts must be recorded. All data should be timestamped.

Analyzing availability data

The instrumentation data must be aggregated and correlated to support the following types of analysis:

- The immediate availability of the system and subsystems.
- The availability failure rates of the system and subsystems. Ideally, an operator should be able to correlate failures with specific activities: what was happening when the system failed?
- A historical view of failure rates of the system or any subsystems across any specified period, and the load on the system (number of user requests, for example) when a failure occurred.
- The reasons for unavailability of the system or any subsystems. For example, the reasons might be service not running, connectivity lost, connected but timing out, and connected but returning errors.

You can calculate the percentage availability of a service over a period of time by using the following formula:

```
%Availability =  ((Total Time – Total Downtime) / Total Time ) * 100
```

This is useful for SLA purposes. (SLA monitoring is described in more detail later in this guidance.) The definition of *downtime* depends on the service. For example, Visual Studio Team Services Build Service defines

downtime as the period (total accumulated minutes) during which Build Service is unavailable. A minute is considered unavailable if all continuous HTTP requests to Build Service to perform customer-initiated operations throughout the minute either result in an error code or do not return a response.

Performance monitoring

As the system is placed under more and more stress (by increasing the volume of users), the size of the datasets that these users access grows and the possibility of failure of one or more components becomes more likely. Frequently, component failure is preceded by a decrease in performance. If you're able detect such a decrease, you can take proactive steps to remedy the situation.

System performance depends on a number of factors. Each factor is typically measured through key performance indicators (KPIs), such as the number of database transactions per second or the volume of network requests that are successfully serviced in a specified time frame. Some of these KPIs might be available as specific performance measures, whereas others might be derived from a combination of metrics.

> **NOTE**
>
> Determining poor or good performance requires that you understand the level of performance at which the system should be capable of running. This requires observing the system while it's functioning under a typical load and capturing the data for each KPI over a period of time. This might involve running the system under a simulated load in a test environment and gathering the appropriate data before deploying the system to a production environment.
>
> You should also ensure that monitoring for performance purposes does not become a burden on the system. You might be able to dynamically adjust the level of detail for the data that the performance monitoring process gathers.

Requirements for performance monitoring

To examine system performance, an operator typically needs to see information that includes:

- The response rates for user requests.
- The number of concurrent user requests.
- The volume of network traffic.
- The rates at which business transactions are being completed.
- The average processing time for requests.

It can also be helpful to provide tools that enable an operator to help spot correlations, such as:

- The number of concurrent users versus request latency times (how long it takes to start processing a request after the user has sent it).
- The number of concurrent users versus the average response time (how long it takes to complete a request after it has started processing).
- The volume of requests versus the number of processing errors.

Along with this high-level functional information, an operator should be able to obtain a detailed view of the performance for each component in the system. This data is typically provided through low-level performance counters that track information such as:

- Memory utilization.
- Number of threads.
- CPU processing time.
- Request queue length.
- Disk or network I/O rates and errors.
- Number of bytes written or read.
- Middleware indicators, such as queue length.

All visualizations should allow an operator to specify a time period. The displayed data might be a snapshot of the current situation and/or a historical view of the performance.

An operator should be able to raise an alert based on any performance measure for any specified value during any specified time interval.

Data sources, instrumentation, and data-collection requirements

You can gather high-level performance data (throughput, number of concurrent users, number of business transactions, error rates, and so on) by monitoring the progress of users' requests as they arrive and pass through the system. This involves incorporating tracing statements at key points in the application code, together with timing information. All faults, exceptions, and warnings should be captured with sufficient data for correlating them with the requests that caused them. The Internet Information Services (IIS) log is another useful source.

If possible, you should also capture performance data for any external systems that the application uses. These external systems might provide their own performance counters or other features for requesting performance data. If this is not possible, record information such as the start time and end time of each request made to an external system, together with the status (success, fail, or warning) of the operation. For example, you can use a stopwatch approach to time requests: start a timer when the request starts and then stop the timer when the request finishes.

Low-level performance data for individual components in a system might be available through features and services such as Windows performance counters and Azure Diagnostics.

Analyzing performance data

Much of the analysis work consists of aggregating performance data by user request type and/or the subsystem or service to which each request is sent. An example of a user request is adding an item to a shopping cart or performing the checkout process in an e-commerce system.

Another common requirement is summarizing performance data in selected percentiles. For example, an operator might determine the response times for 99 percent of requests, 95 percent of requests, and 70 percent of requests. There might be SLA targets or other goals set for each percentile. The ongoing results should be reported in near real time to help detect immediate issues. The results should also be aggregated over the longer time for statistical purposes.

In the case of latency issues affecting performance, an operator should be able to quickly identify the cause of the bottleneck by examining the latency of each step that each request performs. The performance data must therefore provide a means of correlating performance measures for each step to tie them to a specific request.

Depending on the visualization requirements, it might be useful to generate and store a data cube that contains views of the raw data. This data cube can allow complex ad hoc querying and analysis of the performance information.

Security monitoring

All commercial systems that include sensitive data must implement a security structure. The complexity of the security mechanism is usually a function of the sensitivity of the data. In a system that requires users to be authenticated, you should record:

- All sign-in attempts, whether they fail or succeed.
- All operations performed by—and the details of all resources accessed by—an authenticated user.
- When a user ends a session and signs out.

Monitoring might be able to help detect attacks on the system. For example, a large number of failed sign-in attempts might indicate a brute-force attack. An unexpected surge in requests might be the result of a distributed denial-of-service (DDoS) attack. You must be prepared to monitor all requests to all resources

regardless of the source of these requests. A system that has a sign-in vulnerability might accidentally expose resources to the outside world without requiring a user to actually sign in.

Requirements for security monitoring

The most critical aspects of security monitoring should enable an operator to quickly:

- Detect attempted intrusions by an unauthenticated entity.
- Identify attempts by entities to perform operations on data for which they have not been granted access.
- Determine whether the system, or some part of the system, is under attack from outside or inside. (For example, a malicious authenticated user might be attempting to bring the system down.)

To support these requirements, an operator should be notified if:

- One account makes repeated failed sign-in attempts within a specified period.
- One authenticated account repeatedly tries to access a prohibited resource during a specified period.
- A large number of unauthenticated or unauthorized requests occur during a specified period.

The information that's provided to an operator should include the host address of the source for each request. If security violations regularly arise from a particular range of addresses, these hosts might be blocked.

A key part in maintaining the security of a system is being able to quickly detect actions that deviate from the usual pattern. Information such as the number of failed and/or successful sign-in requests can be displayed visually to help detect whether there is a spike in activity at an unusual time. (An example of this activity is users signing in at 3:00 AM and performing a large number of operations when their working day starts at 9:00 AM). This information can also be used to help configure time-based autoscaling. For example, if an operator observes that a large number of users regularly sign in at a particular time of day, the operator can arrange to start additional authentication services to handle the volume of work, and then shut down these additional services when the peak has passed.

Data sources, instrumentation, and data-collection requirements

Security is an all-encompassing aspect of most distributed systems. The pertinent data is likely to be generated at multiple points throughout a system. You should consider adopting a Security Information and Event Management (SIEM) approach to gather the security-related information that results from events raised by the application, network equipment, servers, firewalls, antivirus software, and other intrusion-prevention elements.

Security monitoring can incorporate data from tools that are not part of your application. These tools can include utilities that identify port-scanning activities by external agencies, or network filters that detect attempts to gain unauthenticated access to your application and data.

In all cases, the gathered data must enable an administrator to determine the nature of any attack and take the appropriate countermeasures.

Analyzing security data

A feature of security monitoring is the variety of sources from which the data arises. The different formats and level of detail often require complex analysis of the captured data to tie it together into a coherent thread of information. Apart from the simplest of cases (such as detecting a large number of failed sign-ins, or repeated attempts to gain unauthorized access to critical resources), it might not be possible to perform any complex automated processing of security data. Instead, it might be preferable to write this data, timestamped but otherwise in its original form, to a secure repository to allow for expert manual analysis.

SLA monitoring

Many commercial systems that support paying customers make guarantees about the performance of the system in the form of SLAs. Essentially, SLAs state that the system can handle a defined volume of work within an agreed time frame and without losing critical information. SLA monitoring is concerned with ensuring that

the system can meet measurable SLAs.

> **NOTE**
> SLA monitoring is closely related to performance monitoring. But whereas performance monitoring is concerned with ensuring that the system functions *optimally*, SLA monitoring is governed by a contractual obligation that defines what *optimally* actually means.

SLAs are often defined in terms of:

- Overall system availability. For example, an organization might guarantee that the system will be available for 99.9 percent of the time. This equates to no more than 9 hours of downtime per year, or approximately 10 minutes a week.
- Operational throughput. This aspect is often expressed as one or more high-water marks, such as guaranteeing that the system can support up to 100,000 concurrent user requests or handle 10,000 concurrent business transactions.
- Operational response time. The system might also make guarantees for the rate at which requests are processed. An example is that 99 percent of all business transactions will finish within 2 seconds, and no single transaction will take longer than 10 seconds.

> **NOTE**
> Some contracts for commercial systems might also include SLAs for customer support. An example is that all help-desk requests will elicit a response within five minutes, and that 99 percent of all problems will be fully addressed within 1 working day. Effective issue tracking (described later in this section) is key to meeting SLAs such as these.

Requirements for SLA monitoring

At the highest level, an operator should be able to determine at a glance whether the system is meeting the agreed SLAs or not. And if not, the operator should be able to drill down and examine the underlying factors to determine the reasons for substandard performance.

Typical high-level indicators that can be depicted visually include:

- The percentage of service uptime.
- The application throughput (measured in terms of successful transactions and/or operations per second).
- The number of successful/failing application requests.
- The number of application and system faults, exceptions, and warnings.

All of these indicators should be capable of being filtered by a specified period of time.

A cloud application will likely comprise a number of subsystems and components. An operator should be able to select a high-level indicator and see how it's composed from the health of the underlying elements. For example, if the uptime of the overall system falls below an acceptable value, an operator should be able to zoom in and determine which elements are contributing to this failure.

> **NOTE**
> System uptime needs to be defined carefully. In a system that uses redundancy to ensure maximum availability, individual instances of elements might fail, but the system can remain functional. System uptime as presented by health monitoring should indicate the aggregate uptime of each element and not necessarily whether the system has actually halted. Additionally, failures might be isolated. So even if a specific system is unavailable, the remainder of the system might remain available, although with decreased functionality. (In an e-commerce system, a failure in the system might prevent a customer from placing orders, but the customer might still be able to browse the product catalog.)

For alerting purposes, the system should be able to raise an event if any of the high-level indicators exceed a specified threshold. The lower-level details of the various factors that compose the high-level indicator should be available as contextual data to the alerting system.

Data sources, instrumentation, and data-collection requirements

The raw data that's required to support SLA monitoring is similar to the raw data that's required for performance monitoring, together with some aspects of health and availability monitoring. (See those sections for more details.) You can capture this data by:

- Performing endpoint monitoring.
- Logging exceptions, faults, and warnings.
- Tracing the execution of user requests.
- Monitoring the availability of any third-party services that the system uses.
- Using performance metrics and counters.

All data must be timed and timestamped.

Analyzing SLA data

The instrumentation data must be aggregated to generate a picture of the overall performance of the system. Aggregated data must also support drill-down to enable examination of the performance of the underlying subsystems. For example, you should be able to:

- Calculate the total number of user requests during a specified period and determine the success and failure rate of these requests.
- Combine the response times of user requests to generate an overall view of system response times.
- Analyze the progress of user requests to break down the overall response time of a request into the response times of the individual work items in that request.
- Determine the overall availability of the system as a percentage of uptime for any specific period.
- Analyze the percentage time availability of the individual components and services in the system. This might involve parsing logs that third-party services have generated.

Many commercial systems are required to report real performance figures against agreed SLAs for a specified period, typically a month. This information can be used to calculate credits or other forms of repayments for customers if the SLAs are not met during that period. You can calculate availability for a service by using the technique described in the section Analyzing availability data.

For internal purposes, an organization might also track the number and nature of incidents that caused services to fail. Learning how to resolve these issues quickly, or eliminate them completely, will help to reduce downtime and meet SLAs.

Auditing

Depending on the nature of the application, there might be statutory or other legal regulations that specify requirements for auditing users' operations and recording all data access. Auditing can provide evidence that links customers to specific requests. Nonrepudiation is an important factor in many e-business systems to help maintain trust be between a customer and the organization that's responsible for the application or service.

Requirements for auditing

An analyst must be able to trace the sequence of business operations that users are performing so that you can reconstruct users' actions. This might be necessary simply as a matter of record, or as part of a forensic investigation.

Audit information is highly sensitive. It will likely include data that identifies the users of the system, together with the tasks that they're performing. For this reason, audit information will most likely take the form of reports

that are available only to trusted analysts rather than as an interactive system that supports drill-down of graphical operations. An analyst should be able to generate a range of reports. For example, reports might list all users' activities occurring during a specified time frame, detail the chronology of activity for a single user, or list the sequence of operations performed against one or more resources.

Data sources, instrumentation, and data-collection requirements

The primary sources of information for auditing can include:

- The security system that manages user authentication.
- Trace logs that record user activity.
- Security logs that track all identifiable and unidentifiable network requests.

The format of the audit data and the way in which it's stored might be driven by regulatory requirements. For example, it might not be possible to clean the data in any way. (It must be recorded in its original format.) Access to the repository where it's held must be protected to prevent tampering.

Analyzing audit data

An analyst must be able to access the raw data in its entirety, in its original form. Aside from the requirement to generate common audit reports, the tools for analyzing this data are likely to be specialized and kept external to the system.

Usage monitoring

Usage monitoring tracks how the features and components of an application are used. An operator can use the gathered data to:

- Determine which features are heavily used and determine any potential hotspots in the system. High-traffic elements might benefit from functional partitioning or even replication to spread the load more evenly. An operator can also use this information to ascertain which features are infrequently used and are possible candidates for retirement or replacement in a future version of the system.

- Obtain information about the operational events of the system under normal use. For example, in an e-commerce site, you can record the statistical information about the number of transactions and the volume of customers that are responsible for them. This information can be used for capacity planning as the number of customers grows.

- Detect (possibly indirectly) user satisfaction with the performance or functionality of the system. For example, if a large number of customers in an e-commerce system regularly abandon their shopping carts, this might be due to a problem with the checkout functionality.

- Generate billing information. A commercial application or multitenant service might charge customers for the resources that they use.

- Enforce quotas. If a user in a multitenant system exceeds their paid quota of processing time or resource usage during a specified period, their access can be limited or processing can be throttled.

Requirements for usage monitoring

To examine system usage, an operator typically needs to see information that includes:

- The number of requests that are processed by each subsystem and directed to each resource.
- The work that each user is performing.
- The volume of data storage that each user occupies.
- The resources that each user is accessing.

An operator should also be able to generate graphs. For example, a graph might display the most resource-hungry users, or the most frequently accessed resources or system features.

Data sources, instrumentation, and data-collection requirements

Usage tracking can be performed at a relatively high level. It can note the start and end times of each request and the nature of the request (read, write, and so on, depending on the resource in question). You can obtain this information by:

- Tracing user activity.
- Capturing performance counters that measure the utilization for each resource.
- Monitoring the resource consumption by each user.

For metering purposes, you also need to be able to identify which users are responsible for performing which operations, and the resources that these operations use. The gathered information should be detailed enough to enable accurate billing.

Issue tracking

Customers and other users might report issues if unexpected events or behavior occurs in the system. Issue tracking is concerned with managing these issues, associating them with efforts to resolve any underlying problems in the system, and informing customers of possible resolutions.

Requirements for issue tracking

Operators often perform issue tracking by using a separate system that enables them to record and report the details of problems that users report. These details can include the tasks that the user was trying to perform, symptoms of the problem, the sequence of events, and any error or warning messages that were issued.

Data sources, instrumentation, and data-collection requirements

The initial data source for issue-tracking data is the user who reported the issue in the first place. The user might be able to provide additional data such as:

- A crash dump (if the application includes a component that runs on the user's desktop).
- A screen snapshot.
- The date and time when the error occurred, together with any other environmental information such as the user's location.

This information can be used to help the debugging effort and help construct a backlog for future releases of the software.

Analyzing issue-tracking data

Different users might report the same problem. The issue-tracking system should associate common reports.

The progress of the debugging effort should be recorded against each issue report. When the problem is resolved, the customer can be informed of the solution.

If a user reports an issue that has a known solution in the issue-tracking system, the operator should be able to inform the user of the solution immediately.

Tracing operations and debugging software releases

When a user reports an issue, the user is often only aware of the immediate effect that it has on their operations. The user can only report the results of their own experience back to an operator who is responsible for maintaining the system. These experiences are usually just a visible symptom of one or more fundamental problems. In many cases, an analyst will need to dig through the chronology of the underlying operations to establish the root cause of the problem. This process is called *root cause analysis*.

> **NOTE**
> Root cause analysis might uncover inefficiencies in the design of an application. In these situations, it might be possible to rework the affected elements and deploy them as part of a subsequent release. This process requires careful control, and the updated components should be monitored closely.

Requirements for tracing and debugging

For tracing unexpected events and other problems, it's vital that the monitoring data provides enough information to enable an analyst to trace back to the origins of these issues and reconstruct the sequence of events that occurred. This information must be sufficient to enable an analyst to diagnose the root cause of any problems. A developer can then make the necessary modifications to prevent them from recurring.

Data sources, instrumentation, and data-collection requirements

Troubleshooting can involve tracing all the methods (and their parameters) invoked as part of an operation to build up a tree that depicts the logical flow through the system when a customer makes a specific request. Exceptions and warnings that the system generates as a result of this flow need to be captured and logged.

To support debugging, the system can provide hooks that enable an operator to capture state information at crucial points in the system. Or, the system can deliver detailed step-by-step information as selected operations progress. Capturing data at this level of detail can impose an additional load on the system and should be a temporary process. An operator uses this process mainly when a highly unusual series of events occurs and is difficult to replicate, or when a new release of one or more elements into a system requires careful monitoring to ensure that the elements function as expected.

The monitoring and diagnostics pipeline

Monitoring a large-scale distributed system poses a significant challenge. Each of the scenarios described in the previous section should not necessarily be considered in isolation. There is likely to be a significant overlap in the monitoring and diagnostic data that's required for each situation, although this data might need to be processed and presented in different ways. For these reasons, you should take a holistic view of monitoring and diagnostics.

You can envisage the entire monitoring and diagnostics process as a pipeline that comprises the stages shown in Figure 1.

Figure 1 - The stages in the monitoring and diagnostics pipeline.

Figure 1 highlights how the data for monitoring and diagnostics can come from a variety of data sources. The instrumentation and collection stages are concerned with identifying the sources from where the data needs to be captured, determining which data to capture, how to capture it, and how to format this data so that it can be easily examined. The analysis/diagnosis stage takes the raw data and uses it to generate meaningful information that an operator can use to determine the state of the system. The operator can use this information to make decisions about possible actions to take, and then feed the results back into the instrumentation and collection

stages. The visualization/alerting stage phase presents a consumable view of the system state. It can display information in near real time by using a series of dashboards. And it can generate reports, graphs, and charts to provide a historical view of the data that can help identify long-term trends. If information indicates that a KPI is likely to exceed acceptable bounds, this stage can also trigger an alert to an operator. In some cases, an alert can also be used to trigger an automated process that attempts to take corrective actions, such as autoscaling.

Note that these steps constitute a continuous-flow process where the stages are happening in parallel. Ideally, all the phases should be dynamically configurable. At some points, especially when a system has been newly deployed or is experiencing problems, it might be necessary to gather extended data on a more frequent basis. At other times, it should be possible to revert to capturing a base level of essential information to verify that the system is functioning properly.

Additionally, the entire monitoring process should be considered a live, ongoing solution that's subject to fine-tuning and improvements as a result of feedback. For example, you might start with measuring many factors to determine system health. Analysis over time might lead to a refinement as you discard measures that aren't relevant, enabling you to more precisely focus on the data that you need while minimizing background noise.

Sources of monitoring and diagnostic data

The information that the monitoring process uses can come from several sources, as illustrated in Figure 1. At the application level, information comes from trace logs incorporated into the code of the system. Developers should follow a standard approach for tracking the flow of control through their code. For example, an entry to a method can emit a trace message that specifies the name of the method, the current time, the value of each parameter, and any other pertinent information. Recording the entry and exit times can also prove useful.

You should log all exceptions and warnings, and ensure that you retain a full trace of any nested exceptions and warnings. Ideally, you should also capture information that identifies the user who is running the code, together with activity correlation information (to track requests as they pass through the system). And you should log attempts to access all resources such as message queues, databases, files, and other dependent services. This information can be used for metering and auditing purposes.

Many applications use libraries and frameworks to perform common tasks such as accessing a data store or communicating over a network. These frameworks might be configurable to provide their own trace messages and raw diagnostic information, such as transaction rates and data transmission successes and failures.

> **NOTE**
>
> Many modern frameworks automatically publish performance and trace events. Capturing this information is simply a matter of providing a means to retrieve and store it where it can be processed and analyzed.

The operating system where the application is running can be a source of low-level system-wide information, such as performance counters that indicate I/O rates, memory utilization, and CPU usage. Operating system errors (such as the failure to open a file correctly) might also be reported.

You should also consider the underlying infrastructure and components on which your system runs. Virtual machines, virtual networks, and storage services can all be sources of important infrastructure-level performance counters and other diagnostic data.

If your application uses other external services, such as a web server or database management system, these services might publish their own trace information, logs, and performance counters. Examples include SQL Server Dynamic Management Views for tracking operations performed against a SQL Server database, and IIS trace logs for recording requests made to a web server.

As the components of a system are modified and new versions are deployed, it's important to be able to attribute issues, events, and metrics to each version. This information should be tied back to the release pipeline

so that problems with a specific version of a component can be tracked quickly and rectified.

Security issues might occur at any point in the system. For example, a user might attempt to sign in with an invalid user ID or password. An authenticated user might try to obtain unauthorized access to a resource. Or a user might provide an invalid or outdated key to access encrypted information. Security-related information for successful and failing requests should always be logged.

The section Instrumenting an application contains more guidance on the information that you should capture. But you can use a variety of strategies to gather this information:

- **Application/system monitoring**. This strategy uses internal sources within the application, application frameworks, operating system, and infrastructure. The application code can generate its own monitoring data at notable points during the lifecycle of a client request. The application can include tracing statements that might be selectively enabled or disabled as circumstances dictate. It might also be possible to inject diagnostics dynamically by using a diagnostics framework. These frameworks typically provide plug-ins that can attach to various instrumentation points in your code and capture trace data at these points.

  Additionally, your code and/or the underlying infrastructure might raise events at critical points. Monitoring agents that are configured to listen for these events can record the event information.

- **Real user monitoring**. This approach records the interactions between a user and the application and observes the flow of each request and response. This information can have a two-fold purpose: it can be used for metering usage by each user, and it can be used to determine whether users are receiving a suitable quality of service (for example, fast response times, low latency, and minimal errors). You can use the captured data to identify areas of concern where failures occur most often. You can also use the data to identify elements where the system slows down, possibly due to hotspots in the application or some other form of bottleneck. If you implement this approach carefully, it might be possible to reconstruct users' flows through the application for debugging and testing purposes.

  > **IMPORTANT**
  > You should consider the data that's captured by monitoring real users to be highly sensitive because it might include confidential material. If you save captured data, store it securely. If you want to use the data for performance monitoring or debugging purposes, strip out all personally identifiable information first.

- **Synthetic user monitoring**. In this approach, you write your own test client that simulates a user and performs a configurable but typical series of operations. You can track the performance of the test client to help determine the state of the system. You can also use multiple instances of the test client as part of a load-testing operation to establish how the system responds under stress, and what sort of monitoring output is generated under these conditions.

  > **NOTE**
  > You can implement real and synthetic user monitoring by including code that traces and times the execution of method calls and other critical parts of an application.

- **Profiling**. This approach is primarily targeted at monitoring and improving application performance. Rather than operating at the functional level of real and synthetic user monitoring, it captures lower-level information as the application runs. You can implement profiling by using periodic sampling of the execution state of an application (determining which piece of code that the application is running at a given point in time). You can also use instrumentation that inserts probes into the code at important junctures (such as the start and end of a method call) and records which methods were invoked, at what time, and how long each call took. You can then analyze this data to determine which parts of the

application might cause performance problems.

- **Endpoint monitoring**. This technique uses one or more diagnostic endpoints that the application exposes specifically to enable monitoring. An endpoint provides a pathway into the application code and can return information about the health of the system. Different endpoints can focus on various aspects of the functionality. You can write your own diagnostics client that sends periodic requests to these endpoints and assimilate the responses. For more information, see the Health Endpoint Monitoring pattern.

For maximum coverage, you should use a combination of these techniques.

Instrumenting an application

Instrumentation is a critical part of the monitoring process. You can make meaningful decisions about the performance and health of a system only if you first capture the data that enables you to make these decisions. The information that you gather by using instrumentation should be sufficient to enable you to assess performance, diagnose problems, and make decisions without requiring you to sign in to a remote production server to perform tracing (and debugging) manually. Instrumentation data typically comprises metrics and information that's written to trace logs.

The contents of a trace log can be the result of textual data that's written by the application or binary data that's created as the result of a trace event (if the application is using Event Tracing for Windows--ETW). They can also be generated from system logs that record events arising from parts of the infrastructure, such as a web server. Textual log messages are often designed to be human-readable, but they should also be written in a format that enables an automated system to parse them easily.

You should also categorize logs. Don't write all trace data to a single log, but use separate logs to record the trace output from different operational aspects of the system. You can then quickly filter log messages by reading from the appropriate log rather than having to process a single lengthy file. Never write information that has different security requirements (such as audit information and debugging data) to the same log.

> **NOTE**
>
> A log might be implemented as a file on the file system, or it might be held in some other format, such as a blob in blob storage. Log information might also be held in more structured storage, such as rows in a table.

Metrics will generally be a measure or count of some aspect or resource in the system at a specific time, with one or more associated tags or dimensions (sometimes called a *sample*). A single instance of a metric is usually not useful in isolation. Instead, metrics have to be captured over time. The key issue to consider is which metrics you should record and how frequently. Generating data for metrics too often can impose a significant additional load on the system, whereas capturing metrics infrequently might cause you to miss the circumstances that lead to a significant event. The considerations will vary from metric to metric. For example, CPU utilization on a server might vary significantly from second to second, but high utilization becomes an issue only if it's long-lived over a number of minutes.

Information for correlating data

You can easily monitor individual system-level performance counters, capture metrics for resources, and obtain application trace information from various log files. But some forms of monitoring require the analysis and diagnostics stage in the monitoring pipeline to correlate the data that's retrieved from several sources. This data might take several forms in the raw data, and the analysis process must be provided with sufficient instrumentation data to be able to map these different forms. For example, at the application framework level, a task might be identified by a thread ID. Within an application, the same work might be associated with the user ID for the user who is performing that task.

Also, there's unlikely to be a 1:1 mapping between threads and user requests, because asynchronous operations might reuse the same threads to perform operations on behalf of more than one user. To complicate matters further, a single request might be handled by more than one thread as execution flows through the system. If possible, associate each request with a unique activity ID that's propagated through the system as part of the request context. (The technique for generating and including activity IDs in trace information depends on the technology that's used to capture the trace data.)

All monitoring data should be timestamped in the same way. For consistency, record all dates and times by using Coordinated Universal Time. This will help you more easily trace sequences of events.

> **NOTE**
> Computers operating in different time zones and networks might not be synchronized. Don't depend on using timestamps alone for correlating instrumentation data that spans multiple machines.

Information to include in the instrumentation data

Consider the following points when you're deciding which instrumentation data you need to collect:

- Make sure that information captured by trace events is machine and human readable. Adopt well-defined schemas for this information to facilitate automated processing of log data across systems, and to provide consistency to operations and engineering staff reading the logs. Include environmental information, such as the deployment environment, the machine on which the process is running, the details of the process, and the call stack.
- Enable profiling only when necessary because it can impose a significant overhead on the system. Profiling by using instrumentation records an event (such as a method call) every time it occurs, whereas sampling records only selected events. The selection can be time-based (once every *n* seconds), or frequency-based (once every *n* requests). If events occur very frequently, profiling by instrumentation might cause too much of a burden and itself affect overall performance. In this case, the sampling approach might be preferable. However, if the frequency of events is low, sampling might miss them. In this case, instrumentation might be the better approach.
- Provide sufficient context to enable a developer or administrator to determine the source of each request. This might include some form of activity ID that identifies a specific instance of a request. It might also include information that can be used to correlate this activity with the computational work performed and the resources used. Note that this work might cross process and machine boundaries. For metering, the context should also include (either directly or indirectly via other correlated information) a reference to the customer who caused the request to be made. This context provides valuable information about the application state at the time that the monitoring data was captured.
- Record all requests, and the locations or regions from which these requests are made. This information can assist in determining whether there are any location-specific hotspots. This information can also be useful in determining whether to repartition an application or the data that it uses.
- Record and capture the details of exceptions carefully. Often, critical debug information is lost as a result of poor exception handling. Capture the full details of exceptions that the application throws, including any inner exceptions and other context information. Include the call stack if possible.
- Be consistent in the data that the different elements of your application capture, because this can assist in analyzing events and correlating them with user requests. Consider using a comprehensive and configurable logging package to gather information, rather than depending on developers to adopt the same approach as they implement different parts of the system. Gather data from key performance counters, such as the volume of I/O being performed, network utilization, number of requests, memory use, and CPU utilization. Some infrastructure services might provide their own specific performance counters, such as the number of connections to a database, the rate at which transactions are being

performed, and the number of transactions that succeed or fail. Applications might also define their own specific performance counters.

- Log all calls made to external services, such as database systems, web services, or other system-level services that are part of the infrastructure. Record information about the time taken to perform each call, and the success or failure of the call. If possible, capture information about all retry attempts and failures for any transient errors that occur.

Ensuring compatibility with telemetry systems

In many cases, the information that instrumentation produces is generated as a series of events and passed to a separate telemetry system for processing and analysis. A telemetry system is typically independent of any specific application or technology, but it expects information to follow a specific format that's usually defined by a schema. The schema effectively specifies a contract that defines the data fields and types that the telemetry system can ingest. The schema should be generalized to allow for data arriving from a range of platforms and devices.

A common schema should include fields that are common to all instrumentation events, such as the event name, the event time, the IP address of the sender, and the details that are required for correlating with other events (such as a user ID, a device ID, and an application ID). Remember that any number of devices might raise events, so the schema should not depend on the device type. Additionally, various devices might raise events for the same application; the application might support roaming or some other form of cross-device distribution.

The schema might also include domain fields that are relevant to a particular scenario that's common across different applications. This might be information about exceptions, application start and end events, and success and/or failure of web service API calls. All applications that use the same set of domain fields should emit the same set of events, enabling a set of common reports and analytics to be built.

Finally, a schema might contain custom fields for capturing the details of application-specific events.

Best practices for instrumenting applications

The following list summarizes best practices for instrumenting a distributed application running in the cloud.

- Make logs easy to read and easy to parse. Use structured logging where possible. Be concise and descriptive in log messages.
- In all logs, identify the source and provide context and timing information as each log record is written.
- Use the same time zone and format for all timestamps. This will help to correlate events for operations that span hardware and services running in different geographic regions.
- Categorize logs and write messages to the appropriate log file.
- Do not disclose sensitive information about the system or personal information about users. Scrub this information before it's logged, but ensure that the relevant details are retained. For example, remove the ID and password from any database connection strings, but write the remaining information to the log so that an analyst can determine that the system is accessing the correct database. Log all critical exceptions, but enable the administrator to turn logging on and off for lower levels of exceptions and warnings. Also, capture and log all retry logic information. This data can be useful in monitoring the transient health of the system.
- Trace out of process calls, such as requests to external web services or databases.
- Don't mix log messages with different security requirements in the same log file. For example, don't write debug and audit information to the same log.
- With the exception of auditing events, make sure that all logging calls are fire-and-forget operations that do not block the progress of business operations. Auditing events are exceptional because they are critical to the business and can be classified as a fundamental part of business operations.

- Make sure that logging is extensible and does not have any direct dependencies on a concrete target. For example, rather than writing information by using *System.Diagnostics.Trace*, define an abstract interface (such as *ILogger*) that exposes logging methods and that can be implemented through any appropriate means.
- Make sure that all logging is fail-safe and never triggers any cascading errors. Logging must not throw any exceptions.
- Treat instrumentation as an ongoing iterative process and review logs regularly, not just when there is a problem.

Collecting and storing data

The collection stage of the monitoring process is concerned with retrieving the information that instrumentation generates, formatting this data to make it easier for the analysis/diagnosis stage to consume, and saving the transformed data in reliable storage. The instrumentation data that you gather from different parts of a distributed system can be held in a variety of locations and with varying formats. For example, your application code might generate trace log files and generate application event log data, whereas performance counters that monitor key aspects of the infrastructure that your application uses can be captured through other technologies. Any third-party components and services that your application uses might provide instrumentation information in different formats, by using separate trace files, blob storage, or even a custom data store.

Data collection is often performed through a collection service that can run autonomously from the application that generates the instrumentation data. Figure 2 illustrates an example of this architecture, highlighting the instrumentation data-collection subsystem.

Figure 2 - Collecting instrumentation data.

Note that this is a simplified view. The collection service is not necessarily a single process and might comprise many constituent parts running on different machines, as described in the following sections. Additionally, if the analysis of some telemetry data must be performed quickly (hot analysis, as described in the section Supporting hot, warm, and cold analysis later in this document), local components that operate outside the collection service might perform the analysis tasks immediately. Figure 2 depicts this situation for selected events. After analytical processing, the results can be sent directly to the visualization and alerting subsystem. Data that's subjected to warm or cold analysis is held in storage while it awaits processing.

For Azure applications and services, Azure Diagnostics provides one possible solution for capturing data. Azure Diagnostics gathers data from the following sources for each compute node, aggregates it, and then uploads it to Azure Storage:

- IIS logs

- IIS Failed Request logs
- Windows event logs
- Performance counters
- Crash dumps
- Azure Diagnostics infrastructure logs
- Custom error logs
- .NET EventSource
- Manifest-based ETW

For more information, see the article Azure: Telemetry Basics and Troubleshooting.

Strategies for collecting instrumentation data

Considering the elastic nature of the cloud, and to avoid the necessity of manually retrieving telemetry data from every node in the system, you should arrange for the data to be transferred to a central location and consolidated. In a system that spans multiple datacenters, it might be useful to first collect, consolidate, and store data on a region-by-region basis, and then aggregate the regional data into a single central system.

To optimize the use of bandwidth, you can elect to transfer less urgent data in chunks, as batches. However, the data must not be delayed indefinitely, especially if it contains time-sensitive information.

Pulling and pushing instrumentation data

The instrumentation data-collection subsystem can actively retrieve instrumentation data from the various logs and other sources for each instance of the application (the *pull model*). Or, it can act as a passive receiver that waits for the data to be sent from the components that constitute each instance of the application (the *push model*).

One approach to implementing the pull model is to use monitoring agents that run locally with each instance of the application. A monitoring agent is a separate process that periodically retrieves (pulls) telemetry data collected at the local node and writes this information directly to centralized storage that all instances of the application share. This is the mechanism that Azure Diagnostics implements. Each instance of an Azure web or worker role can be configured to capture diagnostic and other trace information that's stored locally. The monitoring agent that runs alongside each instance copies the specified data to Azure Storage. The article Enabling Diagnostics in Azure Cloud Services and Virtual Machines provides more details on this process. Some elements, such as IIS logs, crash dumps, and custom error logs, are written to blob storage. Data from the Windows event log, ETW events, and performance counters is recorded in table storage. Figure 3 illustrates this mechanism.

Figure 3 - Using a monitoring agent to pull information and write to shared storage.

> **NOTE**
>
> Using a monitoring agent is ideally suited to capturing instrumentation data that's naturally pulled from a data source. An example is information from SQL Server Dynamic Management Views or the length of an Azure Service Bus queue.

It's feasible to use the approach just described to store telemetry data for a small-scale application running on a limited number of nodes in a single location. However, a complex, highly scalable, global cloud application might generate huge volumes of data from hundreds of web and worker roles, database shards, and other services. This flood of data can easily overwhelm the I/O bandwidth available with a single, central location. Therefore, your telemetry solution must be scalable to prevent it from acting as a bottleneck as the system expands. Ideally, your solution should incorporate a degree of redundancy to reduce the risks of losing important monitoring information (such as auditing or billing data) if part of the system fails.

To address these issues, you can implement queuing, as shown in Figure 4. In this architecture, the local monitoring agent (if it can be configured appropriately) or custom data-collection service (if not) posts data to a queue. A separate process running asynchronously (the storage writing service in Figure 4) takes the data in this queue and writes it to shared storage. A message queue is suitable for this scenario because it provides "at least once" semantics that help ensure that queued data will not be lost after it's posted. You can implement the storage writing service by using a separate worker role.

Figure 4 - Using a queue to buffer instrumentation data.

The local data-collection service can add data to a queue immediately after it's received. The queue acts as a buffer, and the storage writing service can retrieve and write the data at its own pace. By default, a queue operates on a first-in, first-out basis. But you can prioritize messages to accelerate them through the queue if they contain data that must be handled more quickly. For more information, see the Priority Queue pattern. Alternatively, you can use different channels (such as Service Bus topics) to direct data to different destinations depending on the form of analytical processing that's required.

For scalability, you can run multiple instances of the storage writing service. If there is a high volume of events, you can use an event hub to dispatch the data to different compute resources for processing and storage.

Consolidating instrumentation data

The instrumentation data that the data-collection service retrieves from a single instance of an application gives a localized view of the health and performance of that instance. To assess the overall health of the system, it's necessary to consolidate some aspects of the data in the local views. You can perform this after the data has been stored, but in some cases, you can also achieve it as the data is collected. Rather than being written directly to shared storage, the instrumentation data can pass through a separate data consolidation service that combines data and acts as a filter and cleanup process. For example, instrumentation data that includes the same correlation information such as an activity ID can be amalgamated. (It's possible that a user starts

performing a business operation on one node and then gets transferred to another node in the event of node failure, or depending on how load balancing is configured.) This process can also detect and remove any duplicated data (always a possibility if the telemetry service uses message queues to push instrumentation data out to storage). Figure 5 illustrates an example of this structure.

Figure 5 - Using a separate service to consolidate and clean up instrumentation data.

Storing instrumentation data

The previous discussions have depicted a rather simplistic view of the way in which instrumentation data is stored. In reality, it can make sense to store the different types of information by using technologies that are most appropriate to the way in which each type is likely to be used.

For example, Azure blob and table storage have some similarities in the way in which they're accessed. But they have limitations in the operations that you can perform by using them, and the granularity of the data that they hold is quite different. If you need to perform more analytical operations or require full-text search capabilities on the data, it might be more appropriate to use data storage that provides capabilities that are optimized for specific types of queries and data access. For example:

- Performance counter data can be stored in a SQL database to enable ad hoc analysis.
- Trace logs might be better stored in Azure Cosmos DB.
- Security information can be written to HDFS.
- Information that requires full-text search can be stored through Elasticsearch (which can also speed searches by using rich indexing).

You can implement an additional service that periodically retrieves the data from shared storage, partitions and filters the data according to its purpose, and then writes it to an appropriate set of data stores as shown in Figure 6. An alternative approach is to include this functionality in the consolidation and cleanup process and write the data directly to these stores as it's retrieved rather than saving it in an intermediate shared storage area. Each approach has its advantages and disadvantages. Implementing a separate partitioning service lessens the load on the consolidation and cleanup service, and it enables at least some of the partitioned data to be regenerated if necessary (depending on how much data is retained in shared storage). However, it consumes additional resources. Also, there might be a delay between the receipt of instrumentation data from each application instance and the conversion of this data into actionable information.

Figure 6 - Partitioning data according to analytical and storage requirements.

The same instrumentation data might be required for more than one purpose. For example, performance counters can be used to provide a historical view of system performance over time. This information might be combined with other usage data to generate customer billing information. In these situations, the same data might be sent to more than one destination, such as a document database that can act as a long-term store for holding billing information, and a multidimensional store for handling complex performance analytics.

You should also consider how urgently the data is required. Data that provides information for alerting must be accessed quickly, so it should be held in fast data storage and indexed or structured to optimize the queries that the alerting system performs. In some cases, it might be necessary for the telemetry service that gathers the data on each node to format and save data locally so that a local instance of the alerting system can quickly notify you of any issues. The same data can be dispatched to the storage writing service shown in the previous diagrams and stored centrally if it's also required for other purposes.

Information that's used for more considered analysis, for reporting, and for spotting historical trends is less urgent and can be stored in a manner that supports data mining and ad hoc queries. For more information, see the section Supporting hot, warm, and cold analysis later in this document.

Log rotation and data retention

Instrumentation can generate considerable volumes of data. This data can be held in several places, starting with the raw log files, trace files, and other information captured at each node to the consolidated, cleaned, and partitioned view of this data held in shared storage. In some cases, after the data has been processed and transferred, the original raw source data can be removed from each node. In other cases, it might be necessary or simply useful to save the raw information. For example, data that's generated for debugging purposes might be best left available in its raw form but can then be discarded quickly after any bugs have been rectified.

Performance data often has a longer life so that it can be used for spotting performance trends and for capacity planning. The consolidated view of this data is usually kept online for a finite period to enable fast access. After that, it can be archived or discarded. Data gathered for metering and billing customers might need to be saved indefinitely. Additionally, regulatory requirements might dictate that information collected for auditing and security purposes also needs to be archived and saved. This data is also sensitive and might need to be encrypted or otherwise protected to prevent tampering. You should never record users' passwords or other information that might be used to commit identity fraud. Such details should be scrubbed from the data before it's stored.

Down-sampling

It's useful to store historical data so you can spot long-term trends. Rather than saving old data in its entirety, it might be possible to down-sample the data to reduce its resolution and save storage costs. As an example, rather than saving minute-by-minute performance indicators, you can consolidate data that's more than a month old to form an hour-by-hour view.

Best practices for collecting and storing logging information

The following list summarizes best practices for capturing and storing logging information:

- The monitoring agent or data-collection service should run as an out-of-process service and should be simple to deploy.
- All output from the monitoring agent or data-collection service should be an agnostic format that's independent of the machine, operating system, or network protocol. For example, emit information in a self-describing format such as JSON, MessagePack, or Protobuf rather than ETL/ETW. Using a standard format enables the system to construct processing pipelines; components that read, transform, and send data in the agreed format can be easily integrated.
- The monitoring and data-collection process must be fail-safe and must not trigger any cascading error conditions.

- In the event of a transient failure in sending information to a data sink, the monitoring agent or data-collection service should be prepared to reorder telemetry data so that the newest information is sent first. (The monitoring agent/data-collection service might elect to drop the older data, or save it locally and transmit it later to catch up, at its own discretion.)

Analyzing data and diagnosing issues

An important part of the monitoring and diagnostics process is analyzing the gathered data to obtain a picture of the overall well-being of the system. You should have defined your own KPIs and performance metrics, and it's important to understand how you can structure the data that has been gathered to meet your analysis requirements. It's also important to understand how the data that's captured in different metrics and log files is correlated, because this information can be key to tracking a sequence of events and help diagnose problems that arise.

As described in the section Consolidating instrumentation data, the data for each part of the system is typically captured locally, but it generally needs to be combined with data generated at other sites that participate in the system. This information requires careful correlation to ensure that data is combined accurately. For example, the usage data for an operation might span a node that hosts a website to which a user connects, a node that runs a separate service accessed as part of this operation, and data storage held on another node. This information needs to be tied together to provide an overall view of the resource and processing usage for the operation. Some preprocessing and filtering of data might occur on the node on which the data is captured, whereas aggregation and formatting are more likely to occur on a central node.

Supporting hot, warm, and cold analysis

Analyzing and reformatting data for visualization, reporting, and alerting purposes can be a complex process that consumes its own set of resources. Some forms of monitoring are time-critical and require immediate analysis of data to be effective. This is known as *hot analysis*. Examples include the analyses that are required for alerting and some aspects of security monitoring (such as detecting an attack on the system). Data that's required for these purposes must be quickly available and structured for efficient processing. In some cases, it might be necessary to move the analysis processing to the individual nodes where the data is held.

Other forms of analysis are less time-critical and might require some computation and aggregation after the raw data has been received. This is called *warm analysis*. Performance analysis often falls into this category. In this case, an isolated, single performance event is unlikely to be statistically significant. (It might be caused by a sudden spike or glitch.) The data from a series of events should provide a more reliable picture of system performance.

Warm analysis can also be used to help diagnose health issues. A health event is typically processed through hot analysis and can raise an alert immediately. An operator should be able to drill into the reasons for the health event by examining the data from the warm path. This data should contain information about the events leading up to the issue that caused the health event.

Some types of monitoring generate more long-term data. This analysis can be performed at a later date, possibly according to a predefined schedule. In some cases, the analysis might need to perform complex filtering of large volumes of data captured over a period of time. This is called *cold analysis*. The key requirement is that the data is stored safely after it has been captured. For example, usage monitoring and auditing require an accurate picture of the state of the system at regular points in time, but this state information does not have to be available for processing immediately after it has been gathered.

An operator can also use cold analysis to provide the data for predictive health analysis. The operator can gather historical information over a specified period and use it in conjunction with the current health data (retrieved from the hot path) to spot trends that might soon cause health issues. In these cases, it might be necessary to raise an alert so that corrective action can be taken.

Correlating data

The data that instrumentation captures can provide a snapshot of the system state, but the purpose of analysis is to make this data actionable. For example:

- What has caused an intense I/O loading at the system level at a specific time?
- Is it the result of a large number of database operations?
- Is this reflected in the database response times, the number of transactions per second, and application response times at the same juncture?

If so, one remedial action that might reduce the load might be to shard the data over more servers. In addition, exceptions can arise as a result of a fault in any level of the system. An exception in one level often triggers another fault in the level above.

For these reasons, you need to be able to correlate the different types of monitoring data at each level to produce an overall view of the state of the system and the applications that are running on it. You can then use this information to make decisions about whether the system is functioning acceptably or not, and determine what can be done to improve the quality of the system.

As described in the section Information for correlating data, you must ensure that the raw instrumentation data includes sufficient context and activity ID information to support the required aggregations for correlating events. Additionally, this data might be held in different formats, and it might be necessary to parse this information to convert it into a standardized format for analysis.

Troubleshooting and diagnosing issues

Diagnosis requires the ability to determine the cause of faults or unexpected behavior, including performing root cause analysis. The information that's required typically includes:

- Detailed information from event logs and traces, either for the entire system or for a specified subsystem during a specified time window.
- Complete stack traces resulting from exceptions and faults of any specified level that occur within the system or a specified subsystem during a specified period.
- Crash dumps for any failed processes either anywhere in the system or for a specified subsystem during a specified time window.
- Activity logs recording the operations that are performed either by all users or for selected users during a specified period.

Analyzing data for troubleshooting purposes often requires a deep technical understanding of the system architecture and the various components that compose the solution. As a result, a large degree of manual intervention is often required to interpret the data, establish the cause of problems, and recommend an appropriate strategy to correct them. It might be appropriate simply to store a copy of this information in its original format and make it available for cold analysis by an expert.

Visualizing data and raising alerts

An important aspect of any monitoring system is the ability to present the data in such a way that an operator can quickly spot any trends or problems. Also important is the ability to quickly inform an operator if a significant event has occurred that might require attention.

Data presentation can take several forms, including visualization by using dashboards, alerting, and reporting.

Visualization by using dashboards

The most common way to visualize data is to use dashboards that can display information as a series of charts, graphs, or some other illustration. These items can be parameterized, and an analyst should be able to select the important parameters (such as the time period) for any specific situation.

Dashboards can be organized hierarchically. Top-level dashboards can give an overall view of each aspect of the

system but enable an operator to drill down to the details. For example, a dashboard that depicts the overall disk I/O for the system should allow an analyst to view the I/O rates for each individual disk to ascertain whether one or more specific devices account for a disproportionate volume of traffic. Ideally, the dashboard should also display related information, such as the source of each request (the user or activity) that's generating this I/O. This information can then be used to determine whether (and how) to spread the load more evenly across devices, and whether the system would perform better if more devices were added.

A dashboard might also use color-coding or some other visual cues to indicate values that appear anomalous or that are outside an expected range. Using the previous example:

- A disk with an I/O rate that's approaching its maximum capacity over an extended period (a hot disk) can be highlighted in red.
- A disk with an I/O rate that periodically runs at its maximum limit over short periods (a warm disk) can be highlighted in yellow.
- A disk that's exhibiting normal usage can be displayed in green.

Note that for a dashboard system to work effectively, it must have the raw data to work with. If you are building your own dashboard system, or using a dashboard developed by another organization, you must understand which instrumentation data you need to collect, at what levels of granularity, and how it should be formatted for the dashboard to consume.

A good dashboard does not only display information, it also enables an analyst to pose ad hoc questions about that information. Some systems provide management tools that an operator can use to perform these tasks and explore the underlying data. Alternatively, depending on the repository that's used to hold this information, it might be possible to query this data directly, or import it into tools such as Microsoft Excel for further analysis and reporting.

> **NOTE**
> You should restrict access to dashboards to authorized personnel, because this information might be commercially sensitive. You should also protect the underlying data for dashboards to prevent users from changing it.

Raising alerts

Alerting is the process of analyzing the monitoring and instrumentation data and generating a notification if a significant event is detected.

Alerting helps ensure that the system remains healthy, responsive, and secure. It's an important part of any system that makes performance, availability, and privacy guarantees to the users where the data might need to be acted on immediately. An operator might need to be notified of the event that triggered the alert. Alerting can also be used to invoke system functions such as autoscaling.

Alerting usually depends on the following instrumentation data:

- **Security events**. If the event logs indicate that repeated authentication and/or authorization failures are occurring, the system might be under attack and an operator should be informed.
- **Performance metrics**. The system must quickly respond if a particular performance metric exceeds a specified threshold.
- **Availability information**. If a fault is detected, it might be necessary to quickly restart one or more subsystems, or fail over to a backup resource. Repeated faults in a subsystem might indicate more serious concerns.

Operators might receive alert information by using many delivery channels such as email, a pager device, or an SMS text message. An alert might also include an indication of how critical a situation is. Many alerting systems support subscriber groups, and all operators who are members of the same group can receive the same set of alerts.

An alerting system should be customizable, and the appropriate values from the underlying instrumentation data can be provided as parameters. This approach enables an operator to filter data and focus on those thresholds or combinations of values that are of interest. Note that in some cases, the raw instrumentation data can be provided to the alerting system. In other situations, it might be more appropriate to supply aggregated data. (For example, an alert can be triggered if the CPU utilization for a node has exceeded 90 percent over the last 10 minutes). The details provided to the alerting system should also include any appropriate summary and context information. This data can help reduce the possibility that false-positive events will trip an alert.

Reporting

Reporting is used to generate an overall view of the system. It might incorporate historical data in addition to current information. Reporting requirements themselves fall into two broad categories: operational reporting and security reporting.

Operational reporting typically includes the following aspects:

- Aggregating statistics that you can use to understand resource utilization of the overall system or specified subsystems during a specified time window.
- Identifying trends in resource usage for the overall system or specified subsystems during a specified period.
- Monitoring the exceptions that have occurred throughout the system or in specified subsystems during a specified period.
- Determining the efficiency of the application in terms of the deployed resources, and understanding whether the volume of resources (and their associated cost) can be reduced without affecting performance unnecessarily.

Security reporting is concerned with tracking customers' use of the system. It can include:

- **Auditing user operations**. This requires recording the individual requests that each user performs, together with dates and times. The data should be structured to enable an administrator to quickly reconstruct the sequence of operations that a user performs over a specified period.
- **Tracking resource use by user**. This requires recording how each request for a user accesses the various resources that compose the system, and for how long. An administrator must be able to use this data to generate a utilization report by user over a specified period, possibly for billing purposes.

In many cases, batch processes can generate reports according to a defined schedule. (Latency is not normally an issue.) But they should also be available for generation on an ad hoc basis if needed. As an example, if you are storing data in a relational database such as Azure SQL Database, you can use a tool such as SQL Server Reporting Services to extract and format data and present it as a set of reports.

Related patterns and guidance

- Autoscaling guidance describes how to decrease management overhead by reducing the need for an operator to continually monitor the performance of a system and make decisions about adding or removing resources.
- Health Endpoint Monitoring pattern describes how to implement functional checks within an application that external tools can access through exposed endpoints at regular intervals.
- Priority Queue pattern shows how to prioritize queued messages so that urgent requests are received and can be processed before less urgent messages.

Next steps

- Azure Monitor overview
- Monitor, diagnose, and troubleshoot Microsoft Azure Storage
- Overview of alerts in Microsoft Azure

- View service health notifications by using the Azure portal
- What is Application Insights?
- Performance diagnostics for Azure virtual machines
- Download and install SQL Server Data Tools (SSDT) for Visual Studio

Retry guidance for Azure services

11/19/2021 • 41 minutes to read • Edit Online

Most Azure services and client SDKs include a retry mechanism. However, these differ because each service has different characteristics and requirements, and so each retry mechanism is tuned to a specific service. This guide summarizes the retry mechanism features for the majority of Azure services, and includes information to help you use, adapt, or extend the retry mechanism for that service.

For general guidance on handling transient faults, and retrying connections and operations against services and resources, see Retry guidance.

The following table summarizes the retry features for the Azure services described in this guidance.

SERVICE	RETRY CAPABILITIES	POLICY CONFIGURATION	SCOPE	TELEMETRY FEATURES
Azure Active Directory	Native in MSAL library	Embedded into MSAL library	Internal	None
Cosmos DB	Native in service	Non-configurable	Global	TraceSource
Data Lake Store	Native in client	Non-configurable	Individual operations	None
Event Hubs	Native in client	Programmatic	Client	None
IoT Hub	Native in client SDK	Programmatic	Client	None
Azure Cache for Redis	Native in client	Programmatic	Client	TextWriter
Search	Native in client	Programmatic	Client	ETW or Custom
Service Bus	Native in client	Programmatic	Namespace Manager, Messaging Factory, and Client	ETW
Service Fabric	Native in client	Programmatic	Client	None
SQL Database with ADO.NET	Polly	Declarative and programmatic	Single statements or blocks of code	Custom
SQL Database with Entity Framework	Native in client	Programmatic	Global per AppDomain	None
SQL Database with Entity Framework Core	Native in client	Programmatic	Global per AppDomain	None
Storage	Native in client	Programmatic	Client and individual operations	TraceSource

> **NOTE**
> For most of the Azure built-in retry mechanisms, there is currently no way apply a different retry policy for different types of error or exception. You should configure a policy that provides the optimum average performance and availability. One way to fine-tune the policy is to analyze log files to determine the type of transient faults that are occurring.

Azure Active Directory

Azure Active Directory (Azure AD) is a comprehensive identity and access management cloud solution that combines core directory services, advanced identity governance, security, and application access management. Azure AD also offers developers an identity management platform to deliver access control to their applications, based on centralized policy and rules.

> **NOTE**
> For retry guidance on Managed Service Identity endpoints, see How to use an Azure VM Managed Service Identity (MSI) for token acquisition.

Retry mechanism

There is a built-in retry mechanism for Azure Active Directory in the Microsoft Authentication Library (MSAL) . To avoid unexpected lockouts, we recommend that third-party libraries and application code do **not** retry failed connections, but allow MSAL to handle retries.

Retry usage guidance

Consider the following guidelines when using Azure Active Directory:

- When possible, use the MSAL library and the built-in support for retries.
- If you are using the REST API for Azure Active Directory, retry the operation if the result code is 429 (Too Many Requests) or an error in the 5xx range. Do not retry for any other errors.
- For 429 errors, only retry after the time indicated in the **Retry-After** header.
- For 5xx errors, use exponential back-off, with the first retry at least 5 seconds after the response.
- Do not retry on errors other than 429 and 5xx.

More information

- Microsoft Authentication Library (MSAL)

Cosmos DB

Cosmos DB is a fully managed multi-model database that supports schemaless JSON data. It offers configurable and reliable performance, native JavaScript transactional processing, and is built for the cloud with elastic scale.

Retry mechanism

The `CosmosClient` class automatically retries failed attempts. To set the number of retries and the maximum wait time, configure CosmosClientOptions. Exceptions that the client raises are either beyond the retry policy or are not transient errors. If Cosmos DB throttles the client, it returns an HTTP 429 error. Check the status code in the `CosmosException` class.

Policy configuration

The following table shows the default settings for the `CosmosClientOptions` class.

SETTING	DEFAULT VALUE	DESCRIPTION
MaxRetryAttemptsOnRateLimitedRequests	9	The maximum number of retries if the request fails because Cosmos DB applied rate limiting on the client.
MaxRetryWaitTimeOnRateLimitedRequests	30	The maximum retry time in seconds for the Azure Cosmos DB service.

Example

```
CosmosClient cosmosClient = new CosmosClient("connection-string", new CosmosClientOptions()
{
    MaxRetryAttemptsOnRateLimitedRequests = 5,
    MaxRetryWaitTimeOnRateLimitedRequests = TimeSpan.FromSeconds(15)
});
```

Telemetry

Retry attempts are logged as unstructured trace messages through a .NET **TraceSource**. You must configure a **TraceListener** to capture the events and write them to a suitable destination log.

For example, if you add the following to your App.config file, traces will be generated in a text file in the same location as the executable:

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="SourceSwitch" value="Verbose"/>
    </switches>
    <sources>
      <source name="DocDBTrace" switchName="SourceSwitch" switchType="System.Diagnostics.SourceSwitch" >
        <listeners>
          <add name="MyTextListener" type="System.Diagnostics.TextWriterTraceListener"
traceOutputOptions="DateTime,ProcessId,ThreadId" initializeData="CosmosDBTrace.txt"></add>
        </listeners>
      </source>
    </sources>
  </system.diagnostics>
</configuration>
```

Event Hubs

Azure Event Hubs is a hyperscale telemetry ingestion service that collects, transforms, and stores millions of events.

Retry mechanism

Retry behavior in the Azure Event Hubs Client Library is controlled by the `RetryPolicy` property on the `EventHubClient` class. The default policy retries with exponential backoff when Azure Event Hub returns a transient `EventHubsException` or an `OperationCanceledException`. Default retry policy for Event Hubs is to retry up to 9 times with an exponential back-off time of up to 30 seconds .

Example

```
EventHubClient client = EventHubClient.CreateFromConnectionString("[event_hub_connection_string]");
client.RetryPolicy = RetryPolicy.Default;
```

More information

.NET Standard client library for Azure Event Hubs

IoT Hub

Azure IoT Hub is a service for connecting, monitoring, and managing devices to develop Internet of Things (IoT) applications.

Retry mechanism

The Azure IoT device SDK can detect errors in the network, protocol, or application. Based on the error type, the SDK checks whether a retry needs to be performed. If the error is *recoverable*, the SDK begins to retry using the configured retry policy.

The default retry policy is *exponential back-off with random jitter*, but it can be configured.

Policy configuration

Policy configuration differs by language. For more details, see IoT Hub retry policy configuration.

More information

- IoT Hub retry policy
- Troubleshoot IoT Hub device disconnection

Azure Cache for Redis

Azure Cache for Redis is a fast data access and low latency cache service based on the popular open-source Redis cache. It is secure, managed by Microsoft, and is accessible from any application in Azure.

The guidance in this section is based on using the StackExchange.Redis client to access the cache. A list of other suitable clients can be found on the Redis website, and these may have different retry mechanisms.

Note that the StackExchange.Redis client uses multiplexing through a single connection. The recommended usage is to create an instance of the client at application startup and use this instance for all operations against the cache. For this reason, the connection to the cache is made only once, and so all of the guidance in this section is related to the retry policy for this initial connection—and not for each operation that accesses the cache.

Retry mechanism

The StackExchange.Redis client uses a connection manager class that is configured through a set of options, including:

- **ConnectRetry**. The number of times a failed connection to the cache will be retried.
- **ReconnectRetryPolicy**. The retry strategy to use.
- **ConnectTimeout**. The maximum waiting time in milliseconds.

Policy configuration

Retry policies are configured programmatically by setting the options for the client before connecting to the cache. This can be done by creating an instance of the **ConfigurationOptions** class, populating its properties, and passing it to the **Connect** method.

The built-in classes support linear (constant) delay and exponential backoff with randomized retry intervals. You can also create a custom retry policy by implementing the **IReconnectRetryPolicy** interface.

The following example configures a retry strategy using exponential backoff.

```
var deltaBackOffInMilliseconds = TimeSpan.FromSeconds(5).TotalMilliseconds;
var maxDeltaBackOffInMilliseconds = TimeSpan.FromSeconds(20).TotalMilliseconds;
var options = new ConfigurationOptions
{
    EndPoints = {"localhost"},
    ConnectRetry = 3,
    ReconnectRetryPolicy = new ExponentialRetry(deltaBackOffInMilliseconds, maxDeltaBackOffInMilliseconds),
    ConnectTimeout = 2000
};
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

Alternatively, you can specify the options as a string, and pass this to the **Connect** method. The **ReconnectRetryPolicy** property cannot be set this way, only through code.

```
var options = "localhost,connectRetry=3,connectTimeout=2000";
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

You can also specify options directly when you connect to the cache.

```
var conn = ConnectionMultiplexer.Connect("redis0:6380,redis1:6380,connectRetry=3");
```

For more information, see Stack Exchange Redis Configuration in the StackExchange.Redis documentation.

The following table shows the default settings for the built-in retry policy.

CONTEXT	SETTING	DEFAULT VALUE (V 1.2.2)	MEANING
ConfigurationOptions	ConnectRetry	3	The number of times to repeat connect attempts during the initial connection operation.
	ConnectTimeout	Maximum 5000 ms plus SyncTimeout	Timeout (ms) for connect operations. Not a delay between retry attempts.
	SyncTimeout	1000	Time (ms) to allow for synchronous operations.
	ReconnectRetryPolicy	LinearRetry 5000 ms	Retry every 5000 ms.

> **NOTE**
> For synchronous operations, `SyncTimeout` can add to the end-to-end latency, but setting the value too low can cause excessive timeouts. See How to troubleshoot Azure Cache for Redis. In general, avoid using synchronous operations, and use asynchronous operations instead. For more information, see Pipelines and Multiplexers.

Retry usage guidance

Consider the following guidelines when using Azure Cache for Redis:

- The StackExchange Redis client manages its own retries, but only when establishing a connection to the cache when the application first starts. You can configure the connection timeout, the number of retry attempts, and the time between retries to establish this connection, but the retry policy does not apply to operations against the cache.
- Instead of using a large number of retry attempts, consider falling back by accessing the original data source instead.

Telemetry

You can collect information about connections (but not other operations) using a **TextWriter**.

```
var writer = new StringWriter();
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);
```

An example of the output this generates is shown below.

```
localhost:6379,connectTimeout=2000,connectRetry=3
1 unique nodes specified
Requesting tie-break from localhost:6379 > __Booksleeve_TieBreak...
Allowing endpoints 00:00:02 to respond...
localhost:6379 faulted: SocketFailure on PING
localhost:6379 failed to nominate (Faulted)
> UnableToResolvePhysicalConnection on GET
No masters detected
localhost:6379: Standalone v2.0.0, master; keep-alive: 00:01:00; int: Connecting; sub: Connecting; not in
use: DidNotRespond
localhost:6379: int ops=0, qu=0, qs=0, qc=1, wr=0, sync=1, socks=2; sub ops=0, qu=0, qs=0, qc=0, wr=0,
socks=2
Circular op-count snapshot; int: 0 (0.00 ops/s; spans 10s); sub: 0 (0.00 ops/s; spans 10s)
Sync timeouts: 0; fire and forget: 0; last heartbeat: -1s ago
resetting failing connections to retry...
retrying; attempts left: 2...
...
```

Examples

The following code example configures a constant (linear) delay between retries when initializing the StackExchange.Redis client. This example shows how to set the configuration using a **ConfigurationOptions** instance.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StackExchange.Redis;

namespace RetryCodeSamples
{
    class CacheRedisCodeSamples
    {
        public async static Task Samples()
        {
            var writer = new StringWriter();
            {
                try
                {
                    var retryTimeInMilliseconds = TimeSpan.FromSeconds(4).TotalMilliseconds; // delay
between retries

                    // Using object-based configuration.
                    var options = new ConfigurationOptions
                                        {
                                            EndPoints = { "localhost" },
                                            ConnectRetry = 3,
                                            ReconnectRetryPolicy = new LinearRetry(retryTimeInMilliseconds)
                                        };
                    ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);

                    // Store a reference to the multiplexer for use in the application.
                }
                catch
                {
                    Console.WriteLine(writer.ToString());
                    throw;
                }
            }
        }
    }
}
```

The next example sets the configuration by specifying the options as a string. The connection timeout is the maximum period of time to wait for a connection to the cache, not the delay between retry attempts. Note that the **ReconnectRetryPolicy** property can only be set by code.

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using StackExchange.Redis;

namespace RetryCodeSamples
{
    class CacheRedisCodeSamples
    {
        public async static Task Samples()
        {
            var writer = new StringWriter();
            {
                try
                {
                    // Using string-based configuration.
                    var options = "localhost,connectRetry=3,connectTimeout=2000";
                    ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(options, writer);

                    // Store a reference to the multiplexer for use in the application.
                }
                catch
                {
                    Console.WriteLine(writer.ToString());
                    throw;
                }
            }
        }
    }
}
```

For more examples, see Configuration on the project website.

More information

- Redis website

Azure Search

Azure Search can be used to add powerful and sophisticated search capabilities to a website or application, quickly and easily tune search results, and construct rich and fine-tuned ranking models.

Retry mechanism

Retry behavior in the Azure Search SDK is controlled by the `SetRetryPolicy` method on the SearchServiceClient and SearchIndexClient classes. The default policy retries with exponential backoff when Azure Search returns a 5xx or 408 (Request Timeout) response.

Telemetry

Trace with ETW or by registering a custom trace provider. For more information, see the AutoRest documentation.

Service Bus

Service Bus is a cloud messaging platform that provides loosely coupled message exchange with improved scale and resiliency for components of an application, whether hosted in the cloud or on-premises.

Retry mechanism

Service Bus implements retries using implementations of the abstract **RetryPolicy** class. The namespace and some of the configuration details depend on which Service Bus client SDK package is used:

PACKAGE	DESCRIPTION	NAMESPACE
Microsoft.Azure.ServiceBus	Azure Service Bus client library for .NET Standard.	`Microsoft.ServiceBus`
WindowsAzure.ServiceBus	This package is the older Service Bus client library. It requires .NET Framework 4.5.2.	`Microsoft.Azure.ServiceBus`

Both versions of the client library provide the following built-in implementations of `RetryPolicy`:

- RetryExponential. Implements exponential backoff.
- NoRetry. Does not perform retries. Use this class when you don't need retries at the Service Bus API level, for example when another process manages retries as part of a batch or multistep operation.

The `RetryPolicy.Default` property returns a default policy of type `RetryExponential`. This policy object has the following settings:

SETTING	DEFAULT VALUE	MEANING
MinimalBackoff	0	Minimum back-off interval. Added to the retry interval computed from `deltaBackoff`.
MaximumBackoff	30 seconds	Maximum back-off interval.
DeltaBackoff	3 seconds	Back-off interval between retries. Multiples of this timespan are used for subsequent retry attempts.
MaxRetryCount	5	The maximum number of retries. (Default value is 10 in the `WindowsAzure.ServiceBus` package.)

In addition, the following property is defined in the older `WindowsAzure.ServiceBus` package:

SETTING	DEFAULT VALUE	MEANING
TerminationTimeBuffer	5 seconds	Retry attempts will be abandoned if the remaining time is less than this value.

Service Bus actions can return a range of exceptions, listed in Service Bus messaging exceptions. Exceptions returned from Service Bus expose the **IsTransient** property that indicates whether the client should retry the operation. The built-in **RetryExponential** policy checks this property before retrying.

If the last exception encountered was **ServerBusyException**, the **RetryExponential** policy adds 10 seconds to the computed retry interval. This value cannot be changed.

Custom implementations could use a combination of the exception type and the **IsTransient** property to provide more fine-grained control over retry actions. For example, you could detect a **QuotaExceededException** and take action to drain the queue before retrying sending a message to it.

The following code sets the retry policy on a Service Bus client using the `Microsoft.Azure.ServiceBus` library:

```
const string QueueName = "queue1";
const string ServiceBusConnectionString = "<your_connection_string>";

var policy = new RetryExponential(
    minimumBackoff: TimeSpan.FromSeconds(10),
    maximumBackoff: TimeSpan.FromSeconds(30),
    maximumRetryCount: 3);
var queueClient = new QueueClient(ServiceBusConnectionString, QueueName, ReceiveMode.PeekLock, policy);
```

The retry policy cannot be set at the individual operation level. It applies to all operations for the client.

Retry usage guidance

Consider the following guidelines when using Service Bus:

- When using the built-in **RetryExponential** implementation, do not implement a fallback operation as the policy reacts to Server Busy exceptions and automatically switches to an appropriate retry mode.
- Service Bus supports a feature called Paired Namespaces that implements automatic failover to a backup queue in a separate namespace if the queue in the primary namespace fails. Messages from the secondary queue can be sent back to the primary queue when it recovers. This feature helps to address transient failures. For more information, see Asynchronous Messaging Patterns and High Availability.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

CONTEXT	EXAMPLE MAXIMUM LATENCY	RETRY POLICY	SETTINGS	HOW IT WORKS
Interactive, UI, or foreground	2 seconds*	Exponential	MinimumBackoff = 0 MaximumBackoff = 30 sec. DeltaBackoff = 300 msec. TimeBuffer = 300 msec. MaxRetryCount = 2	Attempt 1: Delay 0 sec. Attempt 2: Delay ~300 msec. Attempt 3: Delay ~900 msec.
Background or batch	30 seconds	Exponential	MinimumBackoff = 1 MaximumBackoff = 30 sec. DeltaBackoff = 1.75 sec. TimeBuffer = 5 sec. MaxRetryCount = 3	Attempt 1: Delay ~1 sec. Attempt 2: Delay ~3 sec. Attempt 3: Delay ~6 sec. Attempt 4: Delay ~13 sec.

* Not including additional delay that is added if a Server Busy response is received.

Telemetry

Service Bus logs retries as ETW events using an **EventSource**. You must attach an **EventListener** to the event source to capture the events and view them in Performance Viewer, or write them to a suitable destination log. The retry events are of the following form:

```
Microsoft-ServiceBus-Client/RetryPolicyIteration
ThreadID="14,500"
FormattedMessage="[TrackingId:] RetryExponential: Operation Get:https://retry-
tests.servicebus.windows.net/TestQueue/?api-version=2014-05 at iteration 0 is retrying after
00:00:00.1000000 sleep because of Microsoft.ServiceBus.Messaging.MessagingCommunicationException: The remote
name could not be resolved: 'retry-tests.servicebus.windows.net'.TrackingId:6a26f99c-dc6d-422e-8565-
f89fdd0d4fe3, TimeStamp:9/5/2014 10:00:13 PM."
trackingId=""
policyType="RetryExponential"
operation="Get:https://retry-tests.servicebus.windows.net/TestQueue/?api-version=2014-05"
iteration="0"
iterationSleep="00:00:00.1000000"
lastExceptionType="Microsoft.ServiceBus.Messaging.MessagingCommunicationException"
exceptionMessage="The remote name could not be resolved: 'retry-
tests.servicebus.windows.net'.TrackingId:6a26f99c-dc6d-422e-8565-f89fdd0d4fe3,TimeStamp:9/5/2014 10:00:13
PM"
```

Examples

The following code example shows how to set the retry policy for:

- A namespace manager. The policy applies to all operations on that manager, and cannot be overridden for individual operations.
- A messaging factory. The policy applies to all clients created from that factory, and cannot be overridden when creating individual clients.
- An individual messaging client. After a client has been created, you can set the retry policy for that client. The policy applies to all operations on that client.

```
using System;
using System.Threading.Tasks;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;

namespace RetryCodeSamples
{
    class ServiceBusCodeSamples
    {
        private const string connectionString =
            @"Endpoint=sb://[my-namespace].servicebus.windows.net/;
                SharedAccessKeyName=RootManageSharedAccessKey;
                SharedAccessKey=C99.........Mk=";

        public async static Task Samples()
        {
            const string QueueName = "TestQueue";

            ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http;

            var namespaceManager = NamespaceManager.CreateFromConnectionString(connectionString);

            // The namespace manager will have a default exponential policy with 10 retry attempts
            // and a 3 second delay delta.
            // Retry delays will be approximately 0 sec, 3 sec, 9 sec, 25 sec and the fixed 30 sec,
            // with an extra 10 sec added when receiving a ServiceBusyException.

            {
                // Set different values for the retry policy, used for all operations on the namespace
manager.
                namespaceManager.Settings.RetryPolicy =
                    new RetryExponential(
                        minBackoff: TimeSpan.FromSeconds(0),
                        maxBackoff: TimeSpan.FromSeconds(30),
                        maxRetryCount: 3);
```

```
                // Policies cannot be specified on a per-operation basis.
                if (!await namespaceManager.QueueExistsAsync(QueueName))
                {
                    await namespaceManager.CreateQueueAsync(QueueName);
                }
            }

            var messagingFactory = MessagingFactory.Create(
                namespaceManager.Address, namespaceManager.Settings.TokenProvider);
            // The messaging factory will have a default exponential policy with 10 retry attempts
            // and a 3 second delay delta.
            // Retry delays will be approximately 0 sec, 3 sec, 9 sec, 25 sec and the fixed 30 sec,
            // with an extra 10 sec added when receiving a ServiceBusyException.

            {
                // Set different values for the retry policy, used for clients created from it.
                messagingFactory.RetryPolicy =
                    new RetryExponential(
                        minBackoff: TimeSpan.FromSeconds(1),
                        maxBackoff: TimeSpan.FromSeconds(30),
                        maxRetryCount: 3);

                // Policies cannot be specified on a per-operation basis.
                var session = await messagingFactory.AcceptMessageSessionAsync();
            }

            {
                var client = messagingFactory.CreateQueueClient(QueueName);
                // The client inherits the policy from the factory that created it.

                // Set different values for the retry policy on the client.
                client.RetryPolicy =
                    new RetryExponential(
                        minBackoff: TimeSpan.FromSeconds(0.1),
                        maxBackoff: TimeSpan.FromSeconds(30),
                        maxRetryCount: 3);

                // Policies cannot be specified on a per-operation basis.
                var session = await client.AcceptMessageSessionAsync();
            }
        }
    }
}
```

More information

- Asynchronous Messaging Patterns and High Availability

Service Fabric

Distributing reliable services in a Service Fabric cluster guards against most of the potential transient faults discussed in this article. Some transient faults are still possible, however. For example, the naming service might be in the middle of a routing change when it gets a request, causing it to throw an exception. If the same request comes 100 milliseconds later, it will probably succeed.

Internally, Service Fabric manages this kind of transient fault. You can configure some settings by using the `OperationRetrySettings` class while setting up your services. The following code shows an example. In most cases, this should not be necessary, and the default settings will be fine.

```
FabricTransportRemotingSettings transportSettings = new FabricTransportRemotingSettings
{
    OperationTimeout = TimeSpan.FromSeconds(30)
};

var retrySettings = new OperationRetrySettings(TimeSpan.FromSeconds(15), TimeSpan.FromSeconds(1), 5);

var clientFactory = new FabricTransportServiceRemotingClientFactory(transportSettings);

var serviceProxyFactory = new ServiceProxyFactory((c) => clientFactory, retrySettings);

var client = serviceProxyFactory.CreateServiceProxy<ISomeService>(
    new Uri("fabric:/SomeApp/SomeStatefulReliableService"),
    new ServicePartitionKey(0));
```

More information

- Remote exception handling

SQL Database using ADO.NET

SQL Database is a hosted SQL database available in a range of sizes and as both a standard (shared) and premium (non-shared) service.

Retry mechanism

SQL Database has no built-in support for retries when accessed using ADO.NET. However, the return codes from requests can be used to determine why a request failed. For more information about SQL Database throttling, see Azure SQL Database resource limits. For a list of relevant error codes, see SQL error codes for SQL Database client applications.

You can use the Polly library to implement retries for SQL Database. See Transient fault handling with Polly.

Retry usage guidance

Consider the following guidelines when accessing SQL Database using ADO.NET:

- Choose the appropriate service option (shared or premium). A shared instance may suffer longer than usual connection delays and throttling due to the usage by other tenants of the shared server. If more predictable performance and reliable low latency operations are required, consider choosing the premium option.
- Ensure that you perform retries at the appropriate level or scope to avoid non-idempotent operations causing inconsistency in the data. Ideally, all operations should be idempotent so that they can be repeated without causing inconsistency. Where this is not the case, the retry should be performed at a level or scope that allows all related changes to be undone if one operation fails; for example, from within a transactional scope. For more information, see Cloud Service Fundamentals Data Access Layer – Transient Fault Handling.
- A fixed interval strategy is not recommended for use with Azure SQL Database except for interactive scenarios where there are only a few retries at very short intervals. Instead, consider using an exponential back-off strategy for the majority of scenarios.
- Choose a suitable value for the connection and command timeouts when defining connections. Too short a timeout may result in premature failures of connections when the database is busy. Too long a timeout may prevent the retry logic working correctly by waiting too long before detecting a failed connection. The value of the timeout is a component of the end-to-end latency; it is effectively added to the retry delay specified in the retry policy for every retry attempt.
- Close the connection after a certain number of retries, even when using an exponential back off retry logic, and retry the operation on a new connection. Retrying the same operation multiple times on the same connection can be a factor that contributes to connection problems. For an example of this technique, see Cloud Service Fundamentals Data Access Layer – Transient Fault Handling.
- When connection pooling is in use (the default) there is a chance that the same connection will be chosen

from the pool, even after closing and reopening a connection. If this is the case, a technique to resolve it is to call the **ClearPool** method of the **SqlConnection** class to mark the connection as not reusable. However, you should do this only after several connection attempts have failed, and only when encountering the specific class of transient failures such as SQL timeouts (error code -2) related to faulty connections.

- If the data access code uses transactions initiated as **TransactionScope** instances, the retry logic should reopen the connection and initiate a new transaction scope. For this reason, the retryable code block should encompass the entire scope of the transaction.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

CONTEXT	SAMPLE TARGET E2E MAX LATENCY	RETRY STRATEGY	SETTINGS	VALUES	HOW IT WORKS
Interactive, UI, or foreground	2 sec	FixedInterval	Retry count Retry interval First fast retry	3 500 ms true	Attempt 1 - delay 0 sec Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	30 sec	ExponentialBack off	Retry count Min back-off Max back-off Delta back-off First fast retry	5 0 sec 60 sec 2 sec false	Attempt 1 - delay 0 sec Attempt 2 - delay ~2 sec Attempt 3 - delay ~6 sec Attempt 4 - delay ~14 sec Attempt 5 - delay ~30 sec

> **NOTE**
> The end-to-end latency targets assume the default timeout for connections to the service. If you specify longer connection timeouts, the end-to-end latency will be extended by this additional time for every retry attempt.

Examples

This section shows how you can use Polly to access Azure SQL Database using a set of retry policies configured in the `Policy` class.

The following code shows an extension method on the `SqlCommand` class that calls `ExecuteAsync` with exponential backoff.

```
public async static Task<SqlDataReader> ExecuteReaderWithRetryAsync(this SqlCommand command)
{
    GuardConnectionIsNotNull(command);

    var policy = Policy.Handle<Exception>().WaitAndRetryAsync(
        retryCount: 3, // Retry 3 times
        sleepDurationProvider: attempt => TimeSpan.FromMilliseconds(200 * Math.Pow(2, attempt - 1)), //
Exponential backoff based on an initial 200 ms delay.
        onRetry: (exception, attempt) =>
        {
            // Capture some information for logging/telemetry.
            logger.LogWarn($"ExecuteReaderWithRetryAsync: Retry {attempt} due to {exception}.");
        });

    // Retry the following call according to the policy.
    await policy.ExecuteAsync<SqlDataReader>(async token =>
    {
        // This code is executed within the Policy

        if (conn.State != System.Data.ConnectionState.Open) await conn.OpenAsync(token);
        return await command.ExecuteReaderAsync(System.Data.CommandBehavior.Default, token);

    }, cancellationToken);
}
```

This asynchronous extension method can be used as follows.

```
var sqlCommand = sqlConnection.CreateCommand();
sqlCommand.CommandText = "[some query]";

using (var reader = await sqlCommand.ExecuteReaderWithRetryAsync())
{
    // Do something with the values
}
```

More information

- Cloud Service Fundamentals Data Access Layer – Transient Fault Handling

For general guidance on getting the most from SQL Database, see Azure SQL Database performance and elasticity guide.

SQL Database using Entity Framework 6

SQL Database is a hosted SQL database available in a range of sizes and as both a standard (shared) and premium (non-shared) service. Entity Framework is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write.

Retry mechanism

Retry support is provided when accessing SQL Database using Entity Framework 6.0 and higher through a mechanism called Connection resiliency / retry logic. The main features of the retry mechanism are:

- The primary abstraction is the **IDbExecutionStrategy** interface. This interface:
  - Defines synchronous and asynchronous **Execute** methods.
  - Defines classes that can be used directly or can be configured on a database context as a default strategy, mapped to provider name, or mapped to a provider name and server name. When configured on a context, retries occur at the level of individual database operations, of which there might be several for a given context operation.

  - Defines when to retry a failed connection, and how.
- It includes several built-in implementations of the **IDbExecutionStrategy** interface:
  - Default: no retrying.
  - Default for SQL Database (automatic): no retrying, but inspects exceptions and wraps them with suggestion to use the SQL Database strategy.
  - Default for SQL Database: exponential (inherited from base class) plus SQL Database detection logic.
- It implements an exponential back-off strategy that includes randomization.
- The built-in retry classes are stateful and are not thread-safe. However, they can be reused after the current operation is completed.
- If the specified retry count is exceeded, the results are wrapped in a new exception. It does not bubble up the current exception.

Policy configuration

Retry support is provided when accessing SQL Database using Entity Framework 6.0 and higher. Retry policies are configured programmatically. The configuration cannot be changed on a per-operation basis.

When configuring a strategy on the context as the default, you specify a function that creates a new strategy on demand. The following code shows how you can create a retry configuration class that extends the **DbConfiguration** base class.

```
public class BloggingContextConfiguration : DbConfiguration
{
  public BlogConfiguration()
  {
    // Set up the execution strategy for SQL Database (exponential) with 5 retries and 4 sec delay
    this.SetExecutionStrategy(
         "System.Data.SqlClient", () => new SqlAzureExecutionStrategy(5, TimeSpan.FromSeconds(4)));
  }
}
```

You can then specify this as the default retry strategy for all operations using the **SetConfiguration** method of the **DbConfiguration** instance when the application starts. By default, EF will automatically discover and use the configuration class.

```
DbConfiguration.SetConfiguration(new BloggingContextConfiguration());
```

You can specify the retry configuration class for a context by annotating the context class with a **DbConfigurationType** attribute. However, if you have only one configuration class, EF will use it without the need to annotate the context.

```
[DbConfigurationType(typeof(BloggingContextConfiguration))]
public class BloggingContext : DbContext
```

If you need to use different retry strategies for specific operations, or disable retries for specific operations, you can create a configuration class that allows you to suspend or swap strategies by setting a flag in the **CallContext**. The configuration class can use this flag to switch strategies, or disable the strategy you provide and use a default strategy. For more information, see Suspend Execution Strategy (EF6 onwards).

Another technique for using specific retry strategies for individual operations is to create an instance of the required strategy class and supply the desired settings through parameters. You then invoke its **ExecuteAsync** method.

```
var executionStrategy = new SqlAzureExecutionStrategy(5, TimeSpan.FromSeconds(4));
var blogs = await executionStrategy.ExecuteAsync(
    async () =>
    {
        using (var db = new BloggingContext("Blogs"))
        {
            // Acquire some values asynchronously and return them
        }
    },
    new CancellationToken()
);
```

The simplest way to use a **DbConfiguration** class is to locate it in the same assembly as the **DbContext** class. However, this is not appropriate when the same context is required in different scenarios, such as different interactive and background retry strategies. If the different contexts execute in separate AppDomains, you can use the built-in support for specifying configuration classes in the configuration file or set it explicitly using code. If the different contexts must execute in the same AppDomain, a custom solution will be required.

For more information, see Code-Based Configuration (EF6 onwards).

The following table shows the default settings for the built-in retry policy when using EF6.

SETTING	DEFAULT VALUE	MEANING
Policy	Exponential	Exponential back-off.
MaxRetryCount	5	The maximum number of retries.
MaxDelay	30 seconds	The maximum delay between retries. This value does not affect how the series of delays are computed. It only defines an upper bound.
DefaultCoefficient	1 second	The coefficient for the exponential back-off computation. This value cannot be changed.
DefaultRandomFactor	1.1	The multiplier used to add a random delay for each entry. This value cannot be changed.
DefaultExponentialBase	2	The multiplier used to calculate the next delay. This value cannot be changed.

Retry usage guidance

Consider the following guidelines when accessing SQL Database using EF6:

- Choose the appropriate service option (shared or premium). A shared instance may suffer longer than usual connection delays and throttling due to the usage by other tenants of the shared server. If predictable performance and reliable low latency operations are required, consider choosing the premium option.
- A fixed interval strategy is not recommended for use with Azure SQL Database. Instead, use an exponential back-off strategy because the service may be overloaded, and longer delays allow more time for it to recover.
- Choose a suitable value for the connection and command timeouts when defining connections. Base the

timeout on both your business logic design and through testing. You may need to modify this value over time as the volumes of data or the business processes change. Too short a timeout may result in premature failures of connections when the database is busy. Too long a timeout may prevent the retry logic working correctly by waiting too long before detecting a failed connection. The value of the timeout is a component of the end-to-end latency, although you cannot easily determine how many commands will execute when saving the context. You can change the default timeout by setting the **CommandTimeout** property of the **DbContext** instance.

- Entity Framework supports retry configurations defined in configuration files. However, for maximum flexibility on Azure you should consider creating the configuration programmatically within the application. The specific parameters for the retry policies, such as the number of retries and the retry intervals, can be stored in the service configuration file and used at runtime to create the appropriate policies. This allows the settings to be changed without requiring the application to be restarted.

Consider starting with the following settings for retrying operations. You cannot specify the delay between retry attempts (it is fixed and generated as an exponential sequence). You can specify only the maximum values, as shown here; unless you create a custom retry strategy. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

CONTEXT	SAMPLE TARGET E2E MAX LATENCY	RETRY POLICY	SETTINGS	VALUES	HOW IT WORKS
Interactive, UI, or foreground	2 seconds	Exponential	MaxRetryCount MaxDelay	3 750 ms	Attempt 1 - delay 0 sec Attempt 2 - delay 750 ms Attempt 3 – delay 750 ms
Background or batch	30 seconds	Exponential	MaxRetryCount MaxDelay	5 12 seconds	Attempt 1 - delay 0 sec Attempt 2 - delay ~1 sec Attempt 3 - delay ~3 sec Attempt 4 - delay ~7 sec Attempt 5 - delay 12 sec

> **NOTE**
> The end-to-end latency targets assume the default timeout for connections to the service. If you specify longer connection timeouts, the end-to-end latency will be extended by this additional time for every retry attempt.

Examples

The following code example defines a simple data access solution that uses Entity Framework. It sets a specific retry strategy by defining an instance of a class named **BlogConfiguration** that extends **DbConfiguration**.

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Data.Entity.SqlServer;
using System.Threading.Tasks;

namespace RetryCodeSamples
{
    public class BlogConfiguration : DbConfiguration
    {
        public BlogConfiguration()
        {
            // Set up the execution strategy for SQL Database (exponential) with 5 retries and 12 sec delay.
            // These values could be loaded from configuration rather than being hard-coded.
            this.SetExecutionStrategy(
                    "System.Data.SqlClient", () => new SqlAzureExecutionStrategy(5,
TimeSpan.FromSeconds(12)));
        }
    }

    // Specify the configuration type if more than one has been defined.
    // [DbConfigurationType(typeof(BlogConfiguration))]
    public class BloggingContext : DbContext
    {
        // Definition of content goes here.
    }

    class EF6CodeSamples
    {
        public async static Task Samples()
        {
            // Execution strategy configured by DbConfiguration subclass, discovered automatically or
            // or explicitly indicated through configuration or with an attribute. Default is no retries.
            using (var db = new BloggingContext("Blogs"))
            {
                // Add, edit, delete blog items here, then:
                await db.SaveChangesAsync();
            }
        }
    }
}
```

More examples of using the Entity Framework retry mechanism can be found in Connection resiliency / retry logic.

More information

- Azure SQL Database performance and elasticity guide

SQL Database using Entity Framework Core

Entity Framework Core is an object-relational mapper that enables .NET Core developers to work with data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write. This version of Entity Framework was written from the ground up, and doesn't automatically inherit all the features from EF6.x.

Retry mechanism

Retry support is provided when accessing SQL Database using Entity Framework Core through a mechanism called connection resiliency. Connection resiliency was introduced in EF Core 1.1.0.

The primary abstraction is the `IExecutionStrategy` interface. The execution strategy for SQL Server, including SQL Azure, is aware of the exception types that can be retried and has sensible defaults for maximum retries, delay between retries, and so on.

Examples

The following code enables automatic retries when configuring the DbContext object, which represents a session with the database.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder
        .UseSqlServer(
            @"Server=
(localdb)\mssqllocaldb;Database=EFMiscellaneous.ConnectionResiliency;Trusted_Connection=True;",
            options => options.EnableRetryOnFailure());
}
```

The following code shows how to execute a transaction with automatic retries, by using an execution strategy. The transaction is defined in a delegate. If a transient failure occurs, the execution strategy will invoke the delegate again.

```
using (var db = new BloggingContext())
{
    var strategy = db.Database.CreateExecutionStrategy();

    strategy.Execute(() =>
    {
        using (var transaction = db.Database.BeginTransaction())
        {
            db.Blogs.Add(new Blog { Url = "https://blogs.msdn.com/dotnet" });
            db.SaveChanges();

            db.Blogs.Add(new Blog { Url = "https://blogs.msdn.com/visualstudio" });
            db.SaveChanges();

            transaction.Commit();
        }
    });
}
```

Azure Storage

Azure Storage services include blob storage, files, and storage queues.

Blobs, Queues and Files

The ClientOptions Class is the base type for all client option types and exposes various common client options like Diagnostics, Retry, Transport. To provide the client configuration options for connecting to Azure Queue, Blob, and File Storage you must use the corresponding derived type. In the next example, you use the QueueClientOptions class (derived from ClientOptions) to configure a client to connect to Azure Queue Service. The Retry property is the set of options that can be specified to influence how retry attempts are made, and how a failure is eligible to be retried.

```
using System;
using System.Threading;
using Azure.Core;
using Azure.Identity;
using Azure.Storage;
using Azure.Storage.Queues;
using Azure.Storage.Queues.Models;

namespace RetryCodeSamples
{
    class AzureStorageCodeSamples {

        public async static Task Samples() {

                // Provide the client configuration options for connecting to Azure Queue Storage
                 QueueClientOptions queueClientOptions = new QueueClientOptions()
                 {
                     Retry = {
                     Delay = TimeSpan.FromSeconds(2),     //The delay between retry attempts for a fixed
approach or the delay on which to base
                                                          //calculations for a backoff-based approach
                     MaxRetries = 5,                      //The maximum number of retry attempts before
giving up
                     Mode = RetryMode.Exponential,        //The approach to use for calculating retry delays
                     MaxDelay = TimeSpan.FromSeconds(10)  //The maximum permissible delay between retry
attempts
                     },

                     GeoRedundantSecondaryUri = new Uri("https://...")
                     // If the GeoRedundantSecondaryUri property is set, the secondary Uri will be used for
GET or HEAD requests during retries.
                     // If the status of the response from the secondary Uri is a 404, then subsequent
retries for the request will not use the
                     // secondary Uri again, as this indicates that the resource may not have propagated
there yet.
                     // Otherwise, subsequent retries will alternate back and forth between primary and
secondary Uri.
                 };

                 Uri queueServiceUri = new Uri("https://storageaccount.queue.core.windows.net/");
                 string accountName = "Storage account name";
                 string accountKey = "storage account key";

                 // Create a client object for the Queue service, including QueueClientOptions.
                 QueueServiceClient serviceClient = new QueueServiceClient(queueServiceUri, new
DefaultAzureCredential(), queueClientOptions);

                 CancellationTokenSource source = new CancellationTokenSource();
                 CancellationToken cancellationToken = source.Token;

                 // Return an async collection of queues in the storage account.
                 var queues = serviceClient.GetQueuesAsync(QueueTraits.None, null, cancellationToken);
```

Table Support

> **NOTE**
>
> WindowsAzure.Storage Nuget Package has been deprecated. For Azure table support, see Microsoft.Azure.Cosmos.Table Nuget Package

Retry mechanism

Retries occur at the individual REST operation level and are an integral part of the client API implementation. The client storage SDK uses classes that implement the IExtendedRetryPolicy Interface.

The built-in classes provide support for linear (constant delay) and exponential with randomization retry intervals. There is also a no retry policy for use when another process is handling retries at a higher level. However, you can implement your own retry classes if you have specific requirements not provided by the built-in classes.

Alternate retries switch between primary and secondary storage service location if you are using read access geo-redundant storage (RA-GRS) and the result of the request is a retryable error. See Azure Storage Redundancy Options for more information.

Policy configuration

Retry policies are configured programmatically. A typical procedure is to create and populate a **TableRequestOptions**, **BlobRequestOptions**, **FileRequestOptions**, or **QueueRequestOptions** instance.

```
TableRequestOptions interactiveRequestOption = new TableRequestOptions()
{
  RetryPolicy = new LinearRetry(TimeSpan.FromMilliseconds(500), 3),
  // For Read-access geo-redundant storage, use PrimaryThenSecondary.
  // Otherwise set this to PrimaryOnly.
  LocationMode = LocationMode.PrimaryThenSecondary,
  // Maximum execution time based on the business use case.
  MaximumExecutionTime = TimeSpan.FromSeconds(2)
};
```

The request options instance can then be set on the client, and all operations with the client will use the specified request options.

```
client.DefaultRequestOptions = interactiveRequestOption;
var stats = await client.GetServiceStatsAsync();
```

You can override the client request options by passing a populated instance of the request options class as a parameter to operation methods.

```
var stats = await client.GetServiceStatsAsync(interactiveRequestOption, operationContext: null);
```

You use an **OperationContext** instance to specify the code to execute when a retry occurs and when an operation has completed. This code can collect information about the operation for use in logs and telemetry.

```
// Set up notifications for an operation
var context = new OperationContext();
context.ClientRequestID = "some request id";
context.Retrying += (sender, args) =>
{
    // Collect retry information
};
context.RequestCompleted += (sender, args) =>
{
    // Collect operation completion information
};
var stats = await client.GetServiceStatsAsync(null, context);
```

In addition to indicating whether a failure is suitable for retry, the extended retry policies return a **RetryContext** object that indicates the number of retries, the results of the last request, whether the next retry will happen in the primary or secondary location (see table below for details). The properties of the **RetryContext** object can be used to decide if and when to attempt a retry. For more information, see IExtendedRetryPolicy.Evaluate Method.

The following tables show the default settings for the built-in retry policies.

Request options:

SETTING	DEFAULT VALUE	MEANING
MaximumExecutionTime	None	Maximum execution time for the request, including all potential retry attempts. If it is not specified, then the amount of time that a request is permitted to take is unlimited. In other words, the request might stop responding.
ServerTimeout	None	Server timeout interval for the request (value is rounded to seconds). If not specified, it will use the default value for all requests to the server. Usually, the best option is to omit this setting so that the server default is used.
LocationMode	None	If the storage account is created with the Read access geo-redundant storage (RA-GRS) replication option, you can use the location mode to indicate which location should receive the request. For example, if **PrimaryThenSecondary** is specified, requests are always sent to the primary location first. If a request fails, it is sent to the secondary location.
RetryPolicy	ExponentialPolicy	See below for details of each option.

Exponential policy:

SETTING	DEFAULT VALUE	MEANING
maxAttempt	3	Number of retry attempts.
deltaBackoff	4 seconds	Back-off interval between retries. Multiples of this timespan, including a random element, will be used for subsequent retry attempts.
MinBackoff	3 seconds	Added to all retry intervals computed from deltaBackoff. This value cannot be changed.
MaxBackoff	120 seconds	MaxBackoff is used if the computed retry interval is greater than MaxBackoff. This value cannot be changed.

Linear policy:

SETTING	DEFAULT VALUE	MEANING
maxAttempt	3	Number of retry attempts.
deltaBackoff	30 seconds	Back-off interval between retries.

Retry usage guidance

Consider the following guidelines when accessing Azure storage services using the storage client API:

- Use the built-in retry policies from the Microsoft.Azure.Storage.RetryPolicies namespace where they are appropriate for your requirements. In most cases, these policies will be sufficient.
- Use the **ExponentialRetry** policy in batch operations, background tasks, or non-interactive scenarios. In these scenarios, you can typically allow more time for the service to recover—with a consequently increased chance of the operation eventually succeeding.
- Consider specifying the **MaximumExecutionTime** property of the **RequestOptions** parameter to limit the total execution time, but take into account the type and size of the operation when choosing a timeout value.
- If you need to implement a custom retry, avoid creating wrappers around the storage client classes. Instead, use the capabilities to extend the existing policies through the **IExtendedRetryPolicy** interface.
- If you are using read access geo-redundant storage (RA-GRS) you can use the **LocationMode** to specify that retry attempts will access the secondary read-only copy of the store should the primary access fail. However, when using this option you must ensure that your application can work successfully with data that may be stale if the replication from the primary store has not yet completed.

Consider starting with the following settings for retrying operations. These settings are general purpose, and you should monitor the operations and fine-tune the values to suit your own scenario.

CONTEXT	SAMPLE TARGET E2E MAX LATENCY	RETRY POLICY	SETTINGS	VALUES	HOW IT WORKS
Interactive, UI, or foreground	2 seconds	Linear	maxAttempt deltaBackoff	3 500 ms	Attempt 1 - delay 500 ms Attempt 2 - delay 500 ms Attempt 3 - delay 500 ms
Background or batch	30 seconds	Exponential	maxAttempt deltaBackoff	5 4 seconds	Attempt 1 - delay ~3 sec Attempt 2 - delay ~7 sec Attempt 3 - delay ~15 sec

Telemetry

Retry attempts are logged to a **TraceSource**. You must configure a **TraceListener** to capture the events and write them to a suitable destination log. You can use the **TextWriterTraceListener** or **XmlWriterTraceListener** to write the data to a log file, the **EventLogTraceListener** to write to the Windows Event Log, or the **EventProviderTraceListener** to write trace data to the ETW subsystem. You can also configure autoflushing of the buffer, and the verbosity of events that will be logged (for example, Error, Warning, Informational, and Verbose). For more information, see Client-side Logging with the .NET Storage Client Library.

Operations can receive an **OperationContext** instance, which exposes a **Retrying** event that can be used to attach custom telemetry logic. For more information, see OperationContext.Retrying Event.

Examples

The following code example shows how to create two **TableRequestOptions** instances with different retry settings; one for interactive requests and one for background requests. The example then sets these two retry policies on the client so that they apply for all requests, and also sets the interactive strategy on a specific request so that it overrides the default settings applied to the client.

```
using System;
using System.Threading.Tasks;
using Microsoft.Azure.Cosmos.Table;

namespace RetryCodeSamples
{
    class AzureStorageCodeSamples
    {
        private const string connectionString = "UseDevelopmentStorage=true";

        public async static Task Samples()
        {
            var storageAccount = CloudStorageAccount.Parse(connectionString);

            TableRequestOptions interactiveRequestOption = new TableRequestOptions()
            {
                RetryPolicy = new LinearRetry(TimeSpan.FromMilliseconds(500), 3),
                // For Read-access geo-redundant storage, use PrimaryThenSecondary.
                // Otherwise set this to PrimaryOnly.
                LocationMode = LocationMode.PrimaryThenSecondary,
                // Maximum execution time based on the business use case.
                MaximumExecutionTime = TimeSpan.FromSeconds(2)
            };

            TableRequestOptions backgroundRequestOption = new TableRequestOptions()
            {
                // Client has a default exponential retry policy with 4 sec delay and 3 retry attempts
                // Retry delays will be approximately 3 sec, 7 sec, and 15 sec
                MaximumExecutionTime = TimeSpan.FromSeconds(30),
                // PrimaryThenSecondary in case of Read-access geo-redundant storage, else set this to
PrimaryOnly
                LocationMode = LocationMode.PrimaryThenSecondary
            };

            var client = storageAccount.CreateCloudTableClient();
            // Client has a default exponential retry policy with 4 sec delay and 3 retry attempts
            // Retry delays will be approximately 3 sec, 7 sec, and 15 sec
            // ServerTimeout and MaximumExecutionTime are not set

            {
                // Set properties for the client (used on all requests unless overridden)
                // Different exponential policy parameters for background scenarios
                client.DefaultRequestOptions = backgroundRequestOption;
                // Linear policy for interactive scenarios
                client.DefaultRequestOptions = interactiveRequestOption;
            }

            {
                // set properties for a specific request
                var stats = await client.GetServiceStatsAsync(interactiveRequestOption, operationContext:
null);
            }

            {
                // Set up notifications for an operation
                var context = new OperationContext();
                context.ClientRequestID = "some request id";
```

```
                context.Retrying += (sender, args) =>
                {
                    // Collect retry information
                };
                context.RequestCompleted += (sender, args) =>
                {
                    // Collect operation completion information
                };
                var stats = await client.GetServiceStatsAsync(null, context);
            }
        }
    }
}
```

More information

- Azure Storage client Library retry policy recommendations
- Storage Client Library 2.0 – Implementing retry policies

General REST and retry guidelines

Consider the following when accessing Azure or third-party services:

- Use a systematic approach to managing retries, perhaps as reusable code, so that you can apply a consistent methodology across all clients and all solutions.
- Consider using a retry framework such as Polly to manage retries if the target service or client has no built-in retry mechanism. This will help you implement a consistent retry behavior, and it may provide a suitable default retry strategy for the target service. However, you may need to create custom retry code for services that have nonstandard behavior, that do not rely on exceptions to indicate transient failures, or if you want to use a **Retry-Response** reply to manage retry behavior.
- The transient detection logic will depend on the actual client API you use to invoke the REST calls. Some clients, such as the newer **HttpClient** class, will not throw exceptions for completed requests with a non-success HTTP status code.
- The HTTP status code returned from the service can help to indicate whether the failure is transient. You may need to examine the exceptions generated by a client or the retry framework to access the status code or to determine the equivalent exception type. The following HTTP codes typically indicate that a retry is appropriate:
  - 408 Request Timeout
  - 429 Too Many Requests
  - 500 Internal Server Error
  - 502 Bad Gateway
  - 503 Service Unavailable
  - 504 Gateway Timeout
- If you base your retry logic on exceptions, the following typically indicate a transient failure where no connection could be established:
  - WebExceptionStatus.ConnectionClosed
  - WebExceptionStatus.ConnectFailure
  - WebExceptionStatus.Timeout
  - WebExceptionStatus.RequestCanceled
- In the case of a service unavailable status, the service might indicate the appropriate delay before retrying in the **Retry-After** response header or a different custom header. Services might also send

additional information as custom headers, or embedded in the content of the response.

- Do not retry for status codes representing client errors (errors in the 4xx range) except for a 408 Request Timeout and 429 Too Many Requests.
- Thoroughly test your retry strategies and mechanisms under a range of conditions, such as different network states and varying system loadings.

Retry strategies

The following are the typical types of retry strategy intervals:

- **Exponential**. A retry policy that performs a specified number of retries, using a randomized exponential back off approach to determine the interval between retries. For example:

```
var random = new Random();

var delta = (int)((Math.Pow(2.0, currentRetryCount) - 1.0) *
            random.Next((int)(this.deltaBackoff.TotalMilliseconds * 0.8),
            (int)(this.deltaBackoff.TotalMilliseconds * 1.2)));
var interval = (int)Math.Min(checked(this.minBackoff.TotalMilliseconds + delta),
                this.maxBackoff.TotalMilliseconds);
retryInterval = TimeSpan.FromMilliseconds(interval);
```

- **Incremental**. A retry strategy with a specified number of retry attempts and an incremental time interval between retries. For example:

```
retryInterval = TimeSpan.FromMilliseconds(this.initialInterval.TotalMilliseconds +
                (this.increment.TotalMilliseconds * currentRetryCount));
```

- **LinearRetry**. A retry policy that performs a specified number of retries, using a specified fixed time interval between retries. For example:

```
retryInterval = this.deltaBackoff;
```

Transient fault handling with Polly

Polly is a library to programmatically handle retries and circuit breaker strategies. The Polly project is a member of the .NET Foundation. For services where the client does not natively support retries, Polly is a valid alternative and avoids the need to write custom retry code, which can be hard to implement correctly. Polly also provides a way to trace errors when they occur, so that you can log retries.

More information

- connection resiliency
- Data Points - EF Core 1.1

Transient fault handling

11/19/2021 • 19 minutes to read • Edit Online

All applications that communicate with remote services and resources must be sensitive to transient faults. This is especially the case for applications that run in the cloud, where the nature of the environment and connectivity over the Internet means these types of faults are likely to be encountered more often. Transient faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy. These faults are often self-correcting, and if the action is repeated after a suitable delay it is likely to succeed.

This document covers general guidance for transient fault handling. For information about handling transient faults when using Microsoft Azure services, see Azure service-specific retry guidelines.

Why do transient faults occur in the cloud?

Transient faults can occur in any environment, on any platform or operating system, and in any kind of application. In solutions that run on local on-premises infrastructure, the performance and availability of the application and its components is typically maintained through expensive and often underused hardware redundancy, and components and resources are located close to each other. While this approach makes a failure less likely, it can still result in transient faults - and even an outage through unforeseen events such as external power supply or network issues, or other disaster scenarios.

Cloud hosting, including private cloud systems, can offer higher overall availability by using shared resources, redundancy, automatic failover, and dynamic resource allocation across many commodity compute nodes. However, the nature of these environments can mean that transient faults are more likely to occur. There are several reasons for this:

- Many resources in a cloud environment are shared, and access to these resources is subject to throttling in order to protect the resource. Some services will refuse connections when the load rises to a specific level, or a maximum throughput rate is reached, in order to allow processing of existing requests and to maintain performance of the service for all users. Throttling helps to maintain the quality of service for neighbors and other tenants using the shared resource.
- Cloud environments are built using vast numbers of commodity hardware units. They deliver performance by dynamically distributing the load across multiple computing units and infrastructure components, and deliver reliability by automatically recycling or replacing failed units. This dynamic nature means that transient faults and temporary connection failures may occasionally occur.
- There are often more hardware components, including network infrastructure such as routers and load balancers, between the application and the resources and services it uses. This additional infrastructure can occasionally introduce additional connection latency and transient connection faults.
- Network conditions between the client and the server may be variable, especially when communication crosses the Internet. Even in on-premises locations, heavy traffic loads may slow communication and cause intermittent connection failures.

Challenges

Transient faults can have a huge effect on the perceived availability of an application, even if it has been thoroughly tested under all foreseeable circumstances. To ensure that cloud-hosted applications operate reliably, they must be able to respond to the following challenges:

- The application must be able to detect faults when they occur, and determine if these faults are likely to be transient, more long-lasting, or are terminal failures. Different resources are likely to return different responses when a fault occurs, and these responses may also vary depending on the context of the operation; for example, the response for an error when reading from storage may be different from response for an error when writing to storage. Many resources and services have well-documented transient failure contracts. However, where such information is not available, it may be difficult to discover the nature of the fault and whether it is likely to be transient.
- The application must be able to retry the operation if it determines that the fault is likely to be transient and keep track of the number of times the operation was retried.
- The application must use an appropriate strategy for the retries. This strategy specifies the number of times it should retry, the delay between each attempt, and the actions to take after a failed attempt. The appropriate number of attempts and the delay between each one are often difficult to determine, and vary based on the type of resource as well as the current operating conditions of the resource and the application itself.

General guidelines

The following guidelines will help you to design a suitable transient fault handling mechanism for your applications:

- **Determine if there is a built-in retry mechanism:**
  - Many services provide an SDK or client library that contains a transient fault handling mechanism. The retry policy it uses is typically tailored to the nature and requirements of the target service. Alternatively, REST interfaces for services may return information that is useful in determining whether a retry is appropriate, and how long to wait before the next retry attempt.
  - Use the built-in retry mechanism where available, unless you have specific and well-understood requirements that make a different retry behavior more appropriate.
- **Determine if the operation is suitable for retrying**:
  - You should only retry operations where the faults are transient (typically indicated by the nature of the error), and if there is at least some likelihood that the operation will succeed when reattempted. There is no point in reattempting operations that indicate an invalid operation such as a database update to an item that does not exist, or requests to a service or resource that has suffered a fatal error.
  - In general, you should implement retries only where the full impact of this can be determined, and the conditions are well understood and can be validated. If not, leave it to the calling code to implement retries. Remember that the errors returned from resources and services outside your control may evolve over time, and you may need to revisit your transient fault detection logic.
  - When you create services or components, consider implementing error codes and messages that will help clients determine whether they should retry failed operations. In particular, indicate if the client should retry the operation (perhaps by returning an **isTransient** value) and suggest a suitable delay before the next retry attempt. If you build a web service, consider returning custom errors defined within your service contracts. Even though generic clients may not be able to read these, they will be useful when building custom clients.
- **Determine an appropriate retry count and interval:**
  - It is vital to optimize the retry count and the interval to the type of use case. If you do not retry a sufficient number of times, the application will be unable to complete the operation and is likely to experience a failure. If you retry too many times, or with too short an interval between tries, the

application can potentially hold resources such as threads, connections, and memory for long periods, which will adversely affect the health of the application.

- The appropriate values for the time interval and the number of retry attempts depend on the type of operation being attempted. For example, if the operation is part of a user interaction, the interval should be short and only a few retries attempted to avoid making users wait for a response (which holds open connections and can reduce availability for other users). If the operation is part of a long running or critical workflow, where canceling and restarting the process is expensive or time-consuming, it is appropriate to wait longer between attempts and retry more times.
- Determining the appropriate intervals between retries is the most difficult part of designing a successful strategy. Typical strategies use the following types of retry interval:
  - **Exponential back-off**. The application waits a short time before the first retry, and then exponentially increasing time between each subsequent retry. For example, it may retry the operation after 3 seconds, 12 seconds, 30 seconds, and so on.
  - **Incremental intervals**. The application waits a short time before the first retry, and then incrementally increasing time between each subsequent retry. For example, it may retry the operation after 3 seconds, 7 seconds, 13 seconds, and so on.
  - **Regular intervals**. The application waits for the same period of time between each attempt. For example, it may retry the operation every 3 seconds.
  - **Immediate retry**. Sometimes a transient fault is brief, perhaps due to an event such as a network packet collision or a spike in a hardware component. In this case, retrying the operation immediately is appropriate because it may succeed if the fault has cleared in the time it takes the application to assemble and send the next request. However, there should never be more than one immediate retry attempt, and you should switch to alternative strategies, such as exponential back-off or fallback actions, if the immediate retry fails.
  - **Randomization**. Any of the retry strategies listed above may include a randomization to prevent multiple instances of the client sending subsequent retry attempts at the same time. For example, one instance may retry the operation after 3 seconds, 11 seconds, 28 seconds, and so on, while another instance may retry the operation after 4 seconds, 12 seconds, 26 seconds, and so on. Randomization is a useful technique that may be combined with other strategies.
- As a general guideline, use an exponential back-off strategy for background operations, and immediate or regular interval retry strategies for interactive operations. In both cases, you should choose the delay and the retry count so that the maximum latency for all retry attempts is within the required end-to-end latency requirement.
- Take into account the combination of all the factors that contribute to the overall maximum timeout for a retried operation. These factors include the time taken for a failed connection to produce a response (typically set by a timeout value in the client) as well as the delay between retry attempts and the maximum number of retries. The total of all these times can result in long overall operation times, especially when using an exponential delay strategy where the interval between retries grows rapidly after each failure. If a process must meet a specific service level agreement (SLA), the overall operation time, including all timeouts and delays, must be within the limits defined in the SLA.
- Overly aggressive retry strategies, which have intervals that are too short or retries that are too frequent, can have an adverse effect on the target resource or service. This may prevent the resource or service from recovering from its overloaded state, and it will continue to block or

refuse requests. This results in a vicious circle where more and more requests are sent to the resource or service, and consequently its ability to recover is further reduced.

  - Take into account the timeout of the operations when choosing the retry intervals to avoid launching a subsequent attempt immediately (for example, if the timeout period is similar to the retry interval). Also consider if you need to keep the total possible period (the timeout plus the retry intervals) to below a specific total time. Operations that have unusually short or very long timeouts may influence how long to wait, and how often to retry the operation.

  - Use the type of the exception and any data it contains, or the error codes and messages returned from the service, to optimize the interval and the number of retries. For example, some exceptions or error codes (such as the HTTP code 503 Service Unavailable with a Retry-After header in the response) may indicate how long the error might last, or that the service has failed and will not respond to any subsequent attempt.

- **Avoid anti-patterns**:

  - In the vast majority of cases, you should avoid implementations that include duplicated layers of retry code. Avoid designs that include cascading retry mechanisms, or that implement retry at every stage of an operation that involves a hierarchy of requests, unless you have specific requirements that demand this. In these exceptional circumstances, use policies that prevent excessive numbers of retries and delay periods, and make sure you understand the consequences. For example, if one component makes a request to another, which then accesses the target service, and you implement retry with a count of three on both calls there will be nine retry attempts in total against the service. Many services and resources implement a built-in retry mechanism and you should investigate how you can disable or modify this if you need to implement retries at a higher level.

  - Never implement an endless retry mechanism. This is likely to prevent the resource or service recovering from overload situations, and cause throttling and refused connections to continue for a longer period. Use a finite number or retries, or implement a pattern such as Circuit Breaker to allow the service to recover.

  - Never perform an immediate retry more than once.

  - Avoid using a regular retry interval, especially when you have a large number of retry attempts, when accessing services and resources in Azure. The optimum approach is this scenario is an exponential back-off strategy with a circuit-breaking capability.

  - Prevent multiple instances of the same client, or multiple instances of different clients, from sending retries at the same times. If this is likely to occur, introduce randomization into the retry intervals.

- **Test your retry strategy and implementation:**

  - Ensure you fully test your retry strategy implementation under as wide a set of circumstances as possible, especially when both the application and the target resources or services it uses are under extreme load. To check behavior during testing, you can:

    - Inject transient and nontransient faults into the service. For example, send invalid requests or add code that detects test requests and responds with different types of errors. For an example using TestApi, see Fault Injection Testing with TestApi and Introduction to TestApi – Part 5: Managed Code Fault Injection APIs.

    - Create a mock of the resource or service that returns a range of errors that the real service may return. Ensure you cover all the types of error that your retry strategy is designed to detect.

  - Force transient errors to occur by temporarily disabling or overloading the service if it is a custom service that you created and deployed (of course, you should not attempt to overload any shared resources or shared services within Azure).
  - For HTTP-based APIs, consider using the FiddlerCore library in your automated tests to change the outcome of HTTP requests, either by adding extra roundtrip times or by changing the response (such as the HTTP status code, headers, body, or other factors). This enables deterministic testing of a subset of the failure conditions, whether transient faults or other types of failure. For more information, see FiddlerCore. For examples of how to use the library, particularly the **HttpMangler** class, examine the source code for the Azure Storage SDK.
  - Perform high load factor and concurrent tests to ensure that the retry mechanism and strategy works correctly under these conditions, and does not have an adverse effect on the operation of the client or cause cross-contamination between requests.
- **Manage retry policy configurations:**
  - A *retry policy* is a combination of all of the elements of your retry strategy. It defines the detection mechanism that determines whether a fault is likely to be transient, the type of interval to use (such as regular, exponential back-off, and randomization), the actual interval value(s), and the number of times to retry.
  - Retries must be implemented in many places within even the simplest application, and in every layer of more complex applications. Rather than hard-coding the elements of each policy at multiple locations, consider using a central point for storing all the policies. For example, store the values such as the interval and retry count in application configuration files, read them at runtime, and programmatically build the retry policies. This makes it easier to manage the settings, and to modify and fine-tune the values in order to respond to changing requirements and scenarios. However, design the system to store the values rather than rereading a configuration file every time, and ensure suitable defaults are used if the values cannot be obtained from configuration.
  - In an Azure Cloud Services application, consider storing the values that are used to build the retry policies at runtime in the service configuration file so that they can be changed without needing to restart the application.
  - Take advantage of built-in or default retry strategies available in the client APIs you use, but only where they are appropriate for your scenario. These strategies are typically general purpose. In some scenarios they may be all that is required, but in other scenarios they may not offer the full range of options to suit your specific requirements. You must understand how the settings will affect your application through testing to determine the most appropriate values.
- **Log and track transient and nontransient faults:**
  - As part of your retry strategy, include exception handling and other instrumentation that logs when retry attempts are made. While an occasional transient failure and retry are to be expected, and do not indicate a problem, regular and increasing numbers of retries are often an indicator of an issue that may cause a failure, or is currently degrading application performance and availability.
  - Log transient faults as Warning entries rather than Error entries so that monitoring systems do not detect them as application errors that may trigger false alerts.
  - Consider storing a value in your log entries that indicates if the retries were caused by throttling in the service, or by other types of faults such as connection failures, so that you can differentiate them during analysis of the data. An increase in the number of throttling errors is often an indicator of a design flaw in the application or the need to switch to a premium service that offers

dedicated hardware.

  - Consider measuring and logging the overall time taken for operations that include a retry mechanism. This is a good indicator of the overall effect of transient faults on user response times, process latency, and the efficiency of the application use cases. Also log the number of retries occurred in order to understand the factors that contributed to the response time.
  - Consider implementing a telemetry and monitoring system that can raise alerts when the number and rate of failures, the average number of retries, or the overall times taken for operations to succeed, is increasing.

- **Manage operations that continually fail:**
  - There will be circumstances where the operation continues to fail at every attempt, and it is vital to consider how you will handle this situation:
    - Although a retry strategy will define the maximum number of times that an operation should be retried, it does not prevent the application repeating the operation again, with the same number of retries. For example, if an order processing service fails with a fatal error that puts it out of action permanently, the retry strategy may detect a connection timeout and consider it to be a transient fault. The code will retry the operation a specified number of times and then give up. However, when another customer places an order, the operation will be attempted again - even though it is sure to fail every time.
    - To prevent continual retries for operations that continually fail, consider implementing the Circuit Breaker pattern. In this pattern, if the number of failures within a specified time window exceeds the threshold, requests are returned to the caller immediately as errors, without attempting to access the failed resource or service.
    - The application can periodically test the service, on an intermittent basis and with long intervals between requests, to detect when it becomes available. An appropriate interval will depend on the scenario, such as the criticality of the operation and the nature of the service, and might be anything between a few minutes and several hours. At the point where the test succeeds, the application can resume normal operations and pass requests to the newly recovered service.
    - In the meantime, it may be possible to fall back to another instance of the service (perhaps in a different datacenter or application), use a similar service that offers compatible (perhaps simpler) functionality, or perform some alternative operations in the hope that the service will become available soon. For example, it may be appropriate to store requests for the service in a queue or data store and replay them later. Otherwise you might be able to redirect the user to an alternative instance of the application, degrade the performance of the application but still offer acceptable functionality, or just return a message to the user indicating that the application is not available at present.

- **Other considerations**
  - When deciding on the values for the number of retries and the retry intervals for a policy, consider if the operation on the service or resource is part of a long-running or multistep operation. It may be difficult or expensive to compensate all the other operational steps that have already succeeded when one fails. In this case, a very long interval and a large number of retries may be acceptable as long as it does not block other operations by holding or locking scarce resources.
  - Consider if retrying the same operation may cause inconsistencies in data. If some parts of a multistep process are repeated, and the operations are not idempotent, it may result in an inconsistency. For example, an operation that increments a value, if repeated, will produce an invalid result. Repeating an operation that sends a message to a queue may cause an inconsistency

in the message consumer if it cannot detect duplicate messages. To prevent this, ensure that you design each step as an idempotent operation. For more information about idempotency, see Idempotency patterns.

- Consider the scope of the operations that will be retried. For example, it may be easier to implement retry code at a level that encompasses several operations, and retry them all if one fails. However, doing this may result in idempotency issues or unnecessary rollback operations.
- If you choose a retry scope that encompasses several operations, take into account the total latency of all of them when determining the retry intervals, when monitoring the time taken, and before raising alerts for failures.
- Consider how your retry strategy may affect neighbors and other tenants in a shared application, or when using shared resources and services. Aggressive retry policies can cause an increasing number of transient faults to occur for these other users and for applications that share the resources and services. Likewise, your application may be affected by the retry policies implemented by other users of the resources and services. For mission-critical applications, you may decide to use premium services that are not shared. This provides you with much more control over the load and consequent throttling of these resources and services, which can help to justify the additional cost.

More information

- Azure service-specific retry guidelines
- Circuit Breaker pattern
- Compensating Transaction pattern
- Idempotency patterns

Performance tuning a distributed application

11/19/2021 • 2 minutes to read • Edit Online

In this series, we walk through several cloud application scenarios, showing how a development team used load tests and metrics to diagnose performance issues. These articles are based on actual load testing that we performed when developing example applications. The code for each scenario is available on GitHub.

Scenarios:

- Distributed business transaction
- Calling multiple backend services
- Event stream processing

What is performance?

Performance is frequently measured in terms of throughput, response time, and availability. Performance targets should be based on business operations. Customer-facing tasks may have more stringent requirements than operational tasks such as generating reports.

Define a service level objective (SLO) that defines performance targets for each workload. You typically achieve this by breaking a performance target into a set of Key Performance Indicators (KPIs), such as:

- Latency or response time of specific requests
- The number of requests performed per second
- The rate at which the system generates exceptions.

Performance targets should explicitly include a target load. Also, not all users will receive exactly the same level of performance, even when accessing the system simultaneously and performing the same work. So an SLO should be framed in terms of percentiles.

An example SLO for might be: "Client requests will have a response within 500 ms @ P90, at loads up to 25 K requests/second."

Challenges of performance tuning a distributed system

It can be especially challenging to diagnose performance issues in a distributed application. Some of the challenges are:

- A single business transaction or operation typically involves multiple components of the system. It can be hard to get a holistic end-to-end view of a single operation.
- Resource consumption is distributed across multiple nodes. To get a consistent view, you need to aggregate logs and metrics in one place.
- The cloud offers elastic scale. Autoscaling is an important technique for handling spikes in load, but it can also mask underlying issues. Also, it can be hard to know which components need to scale and when.
- Cascading failures can cause failures upstream of the root problem. As a result, the first signal of the problem may appear in a different component than the root cause.

General best practices

Performance tuning is both an art and a science, but it can be made closer to science by taking a systematic

approach. Here are some best practices:

- Enable telemetry to collect metrics. Instrument your code. Follow best practices for monitoring. Use correlated tracing so that you can view all the steps in a transaction.
- Monitor the 90/95/99 percentiles, not just average. The average can mask outliers. The sampling rate for metrics also matters. If the sampling rate is too low, it can hide spikes or outliers that might indicate problems.
- Attack one bottleneck at a time. Form a hypothesis and test it by changing one variable at a time. Removing one bottleneck will often uncover another bottleneck further upstream or downstream.
- Errors and retries can have a large impact on performance. If you see that you are being throttled by backend services, scale out or try to optimize usage (for example by tuning database queries).
- Look for common performance anti-patterns.
- Look for opportunities to parallelize. Two common sources of bottlenecks are message queues and databases. In both cases, sharding can help. For more information, see Horizontal, vertical, and functional data partitioning. Look for hot partitions that might indicate imbalanced read or write loads.

Next steps

Read the performance tuning scenarios

- Distributed business transaction
- Calling multiple backend services
- Event stream processing

Performance tuning scenario: Distributed business transactions

11/19/2021 • 9 minutes to read • Edit Online

This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that we did for a sample application. The application is from the Azure Kubernetes Service (AKS) Baseline for microservices.

This article is part of a series. Read the first part here.

Scenario: A client application initiates a business transaction that involves multiple steps.

This scenario involves a drone delivery application that runs on AKS. Customers use a web app to schedule deliveries by drone. Each transaction requires multiple steps that are performed by separate microservices on the back end:

- The Delivery service manages deliveries.
- The Drone Scheduler service schedules drones for pickup.
- The Package service manages packages.

There are two other services: An Ingestion service that accepts client requests and puts them on a queue for processing, and a Workflow service that coordinates the steps in the workflow.

For more information about this scenario, see Designing a microservices architecture.

Test 1: Baseline

For the first load test, the team created a six-node AKS cluster and deployed three replicas of each microservice. The load test was a step-load test, starting at two simulated users and ramping up to 40 simulated users.

SETTING	VALUE
Cluster nodes	6
Pods	3 per service

The following graph shows the results of the load test, as shown in Visual Studio. The purple line plots user load, and the orange line plots total requests.

The first thing to realize about this scenario is that client requests per second is not a useful metric of performance. That's because the application processes requests asynchronously, so the client gets a response right away. The response code is always HTTP 202 (Accepted), meaning the request was accepted but processing is not complete.

What we really want to know is whether the backend is keeping up with the request rate. The Service Bus queue can absorb spikes, but if the backend cannot handle a sustained load, processing will fall further and further behind.

Here's a more informative graph. It plots the number incoming and outgoing messages on the Service Bus queue. Incoming messages are shown in light blue, and outgoing messages are shown in dark blue:

This chart is showing that the rate of incoming messages increases, reaching a peak and then dropping back to zero at the end of the load test. But the number of outgoing messages peaks early in the test and then actually drops. That means the Workflow service, which handles the requests, isn't keeping up. Even after the load test ends (around 9:22 on the graph), messages are still being processed as the Workflow service continues to drain the queue.

What's slowing down the processing? The first thing to look for is errors or exceptions that might indicate a systematic issue. The Application Map in Azure Monitor shows the graph of calls between components, and is a quick way to spot issues and then click through to get more details.

Sure enough, the Application Map shows that the Workflow service is getting errors from the Delivery service:

To see more details, you can select a node in the graph and click into an end-to-end transaction view. In this case, it shows that the Delivery service is returning HTTP 500 errors. The error messages indicate that an exception is being thrown due to memory limits in Azure Cache for Redis.

You may notice that these calls to Redis don't appear in the Application Map. That's because the .NET library for Application Insights doesn't have built-in support for tracking Redis as a dependency. (For a list of what's supported out of the box, see Dependency auto-collection.) As a fallback, you can use the TrackDependency API to track any dependency. Load testing often reveals these kinds of gaps in the telemetry, which can be remediated.

Test 2: Increased cache size

For the second load test, the development team increased the cache size in Azure Cache for Redis. (See How to Scale Azure Cache for Redis.) This change resolved the out-of-memory exceptions, and now the Application Map shows zero errors:

However, there is still a dramatic lag in processing messages. At the peak of the load test, the incoming message rate is more than 5× the outgoing rate:

The following graph measures throughput in terms of message completion — that is, the rate at which the Workflow service marks the Service Bus messages as completed. Each point on the graph represents 5 seconds of data, showing ~16/sec maximum throughput.

This graph was generated by running a query in the Log Analytics workspace, using the Kusto query language:

```
let start=datetime("2020-07-31T22:30:00.000Z");
let end=datetime("2020-07-31T22:45:00.000Z");
dependencies
| where cloud_RoleName == 'fabrikam-workflow'
| where timestamp > start and timestamp < end
| where type == 'Azure Service Bus'
| where target has 'https://dev-i-iuosnlbwkzkau.servicebus.windows.net'
| where client_Type == "PC"
| where name == "Complete"
| summarize succeeded=sumif(itemCount, success == true), failed=sumif(itemCount, success == false) by
bin(timestamp, 5s)
| render timechart
```

Test 3: Scale out the backend services

It appears the back end is the bottleneck. An easy next step is to scale out the business services (Package, Delivery, and Drone Scheduler), and see if throughput improves. For the next load test, the team scaled these services out from three replicas to six replicas.

SETTING	VALUE
Cluster nodes	6
Ingestion service	3 replicas
Workflow service	3 replicas
Package, Delivery, Drone Scheduler services	6 replicas each

Unfortunately this load test shows only modest improvement. Outgoing messages are still not keeping up with incoming messages:

Throughput is more consistent, but the maximum achieved is about the same as the previous test:

Moreover, looking at Azure Monitor for containers, it appears the problem is not caused by resource exhaustion within the cluster. First, the node-level metrics show that CPU utilization remains under 40% even at the 95th percentile, and memory utilization is about 20%.

In a Kubernetes environment, it's possible for individual pods to be resource-constrained even when the nodes aren't. But the pod-level view shows that all pods are healthy.

From this test, it seems that just adding more pods to the back end won't help. The next step is to look more closely at the Workflow service to understand what's happening when it processes messages. Application Insights shows that the average duration of the Workflow service's `Process` operation is 246 ms.

We can also run a query to get metrics on the individual operations within each transaction:

TARGET	PERCENTILE_DURATION_50	PERCENTILE_DURATION_95
`https://dev-i-iuosnlbwkzkau.servicebus.windows.net/ | dev-i-iuosnlbwkzkau`	86.66950203	283.4255578
delivery	37	57
package	12	17
dronescheduler	21	41

The first row in this table represents the Service Bus queue. The other rows are the calls to the backend services. For reference, here's the Log Analytics query for this table:

```
let start=datetime("2020-07-31T22:30:00.000Z");
let end=datetime("2020-07-31T22:45:00.000Z");
let dataset=dependencies
| where timestamp > start and timestamp < end
| where (cloud_RoleName == 'fabrikam-workflow')
| where name == 'Complete' or target in ('package', 'delivery', 'dronescheduler');
dataset
| summarize percentiles(duration, 50, 95) by target
```

```
let start=datetime("2019-08-01T00:00:00.000Z");
let end=datetime("2019-08-01T00:30:00.000Z");
let dataset=dependencies
| where timestamp > start and timestamp < end
| where (cloud_RoleName == 'fabrikam-workflow')
| where name == 'Complete' or target in ('package', 'delivery', 'drone scheduler');
dataset
| summarize percentiles(duration, 50, 95) by target
```

Completed 00:00:03.200 4 records

TABLE CHART Columns

Drag a column header and drop it here to group by that column

target	percentile_duration_50	percentile_duration_95
delivery	37	57
https://dev-i-iuosnlbwkzkau.servicebus.windows.net/ | dev-i-iuosnlbwkzkau	86.6695020315	283.4255577965
package	12	17
dronescheduler	21	41

These latencies look reasonable. But here is the key insight: If the total operation time is ~250 ms, that puts a strict upper bound on how fast messages can be processed in serial. The key to improving throughput, therefore, is greater parallelism.

That should be possible in this scenario, for two reasons:

- These are network calls, so most of the time is spent waiting for I/O completion
- The messages are independent, and don't need to be processed in order.

Test 4: Increase parallelism

For this test, the team focused on increasing parallelism. To do so, they adjusted two settings on the Service Bus client used by the Workflow service:

SETTING	DESCRIPTION	DEFAULT	NEW VALUE
`MaxConcurrentCalls`	The maximum number of messages to process concurrently.	1	20
`PrefetchCount`	How many messages the client will fetch ahead of time into its local cache.	0	3000

For more information about these settings, see Best Practices for performance improvements using Service Bus Messaging. Running the test with these settings produced the following graph:

Recall that incoming messages are shown in light blue, and outgoing messages are shown in dark blue.

At first glance, this is a very weird graph. For a while, the outgoing message rate exactly tracks the incoming rate. But then, at about the 2:03 mark, the rate of incoming messages levels off, while the number of outgoing messages continues to rise, actually exceeding the total number of incoming messages. That seems impossible.

The clue to this mystery can be found in the **Dependencies** view in Application Insights. This chart summarizes all of the calls that the Workflow service made to Service Bus:

Notice that entry for `DeadLetter`. That calls indicates messages are going into the Service Bus dead-letter queue.

To understand what's happening, you need to understand Peek-Lock semantics in Service Bus. When a client uses Peek-Lock, Service Bus atomically retrieves and locks a message. While the lock is held, the message is guaranteed not to be delivered to other receivers. If the lock expires, the message becomes available to other receivers. After a maximum number of delivery attempts (which is configurable), Service Bus will put the messages onto a dead-letter queue, where it can be examined later.

Remember that the Workflow service is prefetching large batches of messages — 3000 messages at a time).

That means the total time to process each message is longer, which results in messages timing out, going back onto the queue, and eventually going into the dead-letter queue.

You can also see this behavior in the exceptions, where numerous `MessageLostLockException` exceptions are recorded:

Test 5: Increase lock duration

For this load test, the message lock duration was set to 5 minutes, to prevent lock timeouts. The graph of incoming and outgoing messages now shows that the system is keeping up with the rate of incoming messages:

Over the total duration of the 8-minute load test, the application completed 25 K operations, with a peak throughput of 72 operations/sec, representing a 400% increase in maximum throughput.

However, running the same test with a longer duration showed that the application could not sustain this rate:

The container metrics show that maximum CPU utilization was close to 100%. At this point, the application appears to be CPU-bound. Scaling the cluster might improve performance now, unlike the previous attempt at scaling out.

Test 6: Scale out the backend services (again)

For the final load test in the series, the team scaled out the Kubernetes cluster and pods as follows:

SETTING	VALUE
Cluster nodes	12
Ingestion service	3 replicas
Workflow service	6 replicas
Package, Delivery, Drone Scheduler services	9 replicas each

This test resulted in a higher sustained throughput, with no significant lags in processing messages. Moreover, node CPU utilization stayed below 80%.

Summary

For this scenario, the following bottlenecks were identified:

- Out-of-memory exceptions in Azure Cache for Redis.
- Lack of parallelism in message processing.
- Insufficient message lock duration, leading to lock timeouts and messages being placed in the dead letter queue.
- CPU exhaustion.

To diagnose these issues, the development team relied on the following metrics:

- The rate of incoming and outgoing Service Bus messages.
- Application Map in Application Insights.
- Errors and exceptions.
- Custom Log Analytics queries.
- CPU and memory utilization in Azure Monitor for containers.

Next steps

For more information about the design of this scenario, see Designing a microservices architecture.

Performance tuning scenario: Multiple backend services

11/19/2021 • 10 minutes to read • Edit Online

This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that was did for a sample application. The application is from the Azure Kubernetes Service (AKS) Baseline for microservices, along with a Visual Studio load test project used to generate the results.

This article is part of a series. Read the first part here.

Scenario: Call multiple backend services to retrieve information and then aggregate the results.

This scenario involves a drone delivery application. Clients can query a REST API to get their latest invoice information. The invoice includes a summary of the customer's deliveries, packages, and total drone utilization. This application uses a microservices architecture running on AKS, and the information needed for the invoice is spread across several microservices.

Rather than the client calling each service directly, the application implements the Gateway Aggregation pattern. Using this pattern, the client makes a single request to a gateway service. The gateway in turn calls the backend services in parallel, and then aggregates the results into a single response payload.

Test 1: Baseline performance

To establish a baseline, the development team started with a step-load test, ramping the load from one simulated user up to 40 users, over a total duration of 8 minutes. The following chart, taken from Visual Studio, shows the results. The purple line shows the user load, and the orange line shows throughput (average requests per second).

The red line along the bottom of the chart shows that no errors were returned to the client, which is encouraging. However, the average throughput peaks about half way through the test, and then drops off for the remainder, even while the load continues to increase. That indicates the back end is not able to keep up. The pattern seen here is common when a system starts to reach resource limits — after reaching a maximum, throughput actually falls significantly. Resource contention, transient errors, or an increase in the rate of exceptions can all contribute to this pattern.

Let's dig into the monitoring data to learn what's happening inside the system. The next chart is taken from Application Insights. It shows the average durations of the HTTP calls from the gateway to the backend services.

This chart shows that one operation in particular, `GetDroneUtilization`, takes much longer on average — by an order of magnitude. The gateway makes these calls in parallel, so the slowest operation determines how long it takes for the entire request to complete.

Clearly the next step is dig into the `GetDroneUtilization` operation and look for any bottlenecks. One possibility is resource exhaustion. Perhaps this particular backend service is running out of CPU or memory. For an AKS cluster, this information is available in the Azure portal through the Azure Monitor for containers feature. The following graphs show resource utilization at the cluster level:

In this screenshot, both the average and maximum values are shown. It's important to look at more than just the average, because the average can hide spikes in the data. Here, the average CPU utilization stays below 50%, but there are a couple of spikes to 80%. That's close to capacity but still within tolerances. Something else is causing the bottleneck.

The next chart reveals the true culprit. This chart shows HTTP response codes from the Delivery service's backend database, which in this case is Cosmos DB. The blue line represents success codes (HTTP 2xx), while the green line represents HTTP 429 errors. An HTTP 429 return code means that Cosmos DB is temporarily throttling requests, because the caller is consuming more resource units (RU) than provisioned.

To get further insight, the development team used Application Insights to view the end-to-end telemetry for a representative sample of requests. Here is one instance:

This view shows the calls related to a single client request, along with timing information and response codes. The top-level calls are from the gateway to the backend services. The call to `GetDroneUtilization` is expanded to show calls to external dependencies — in this case, to Cosmos DB. The call in red returned an HTTP 429 error.

Note the large gap between the HTTP 429 error and the next call. When the Cosmos DB client library receives an HTTP 429 error, it automatically backs off and waits to retry the operation. What this view shows is that during the 672 ms this operation took, most of that time was spent waiting to retry Cosmos DB.

Here's another interesting graph for this analysis. It shows RU consumption per physical partition versus provisioned RUs per physical partition:

To make sense of this graph, you need to understand how Cosmos DB manages partitions. Collections in Cosmos DB can have a *partition key*. Each possible key value defines a logical partition of the data within the

collection. Cosmos DB distributes these logical partitions across one or more *physical* partitions. The management of physical partitions is handled automatically by Cosmos DB. As you store more data, Cosmos DB might move logical partitions into new physical partitions, in order to spread load across the physical partitions.

For this load test, the Cosmos DB collection was provisioned with 900 RUs. The chart shows 100 RU per physical partition, which implies a total of nine physical partitions. Although Cosmos DB automatically handles the sharding of physical partitions, knowing the partition count can give insight into performance. The development team will use this information later, as they continue to optimize. Where the blue line crosses the purple horizontal line, RU consumption has exceeded the provisioned RUs. That's the point where Cosmos DB will begin to throttle calls.

Test 2: Increase resource units

For the second load test, the team scaled out the Cosmos DB collection from 900 RU to 2500 RU. Throughput increased from 19 requests/second to 23 requests/second, and average latency dropped from 669 ms to 569 ms.

METRIC	TEST 1	TEST 2
Throughput (req/sec)	19	23
Average latency (ms)	669	569
Successful requests	9.8 K	11 K

These aren't huge gains, but looking at the graph over time shows a more complete picture:

Whereas the previous test showed an initial spike followed by a sharp drop, this test shows more consistent throughput. However, the maximum throughput is not significantly higher.

All requests to Cosmos DB returned a 2xx status, and the HTTP 429 errors went away:

The graph of RU consumption versus provisioned RUs shows there is plenty of headroom. There are about 275 RUs per physical partition, and the load test peaked at about 100 RUs consumed per second.

Another interesting metric is the number of calls to Cosmos DB per successful operation:

METRIC	TEST 1	TEST 2
Calls per operation	11	9

Assuming no errors, the number of calls should match the actual query plan. In this case, the operation involves a cross-partition query that hits all nine physical partitions. The higher value in the first load test reflects the

number of calls that returned a 429 error.

This metric was calculated by running a custom Log Analytics query:

```
let start=datetime("2020-06-18T20:59:00.000Z");
let end=datetime("2020-07-24T21:10:00.000Z");
let operationNameToEval="GET DroneDeliveries/GetDroneUtilization";
let dependencyType="Azure DocumentDB";
let dataset=requests
| where timestamp > start and timestamp < end
| where success == true
| where name == operationNameToEval;
dataset
| project reqOk=itemCount
| summarize
    SuccessRequests=sum(reqOk),
    TotalNumberOfDepCalls=(toscalar(dependencies
    | where timestamp > start and timestamp < end
    | where type == dependencyType
    | summarize sum(itemCount)))
| project
    OperationName=operationNameToEval,
    DependencyName=dependencyType,
    SuccessRequests,
    AverageNumberOfDepCallsPerOperation=(TotalNumberOfDepCalls/SuccessRequests)
```

To summarize, the second load test shows improvement. However, the `GetDroneUtilization` operation still takes about an order of magnitude longer than the next-slowest operation. Looking at the end-to-end transactions helps to explain why:

As mentioned earlier, the `GetDroneUtilization` operation involves a cross-partition query to Cosmos DB. This means the Cosmos DB client has to fan out the query to each physical partition and collect the results. As the end-to-end transaction view shows, these queries are being performed in serial. The operation takes as long as the sum of all the queries — and this problem will only get worse as the size of the data grows and more physical partitions are added.

Test 3: Parallel queries

Based on the previous results, an obvious way to reduce latency is to issue the queries in parallel. The Cosmos DB client SDK has a setting that controls the maximum degree of parallelism.

VALUE	DESCRIPTION
0	No parallelism (default)
> 0	Maximum number of parallel calls
-1	The client SDK selects an optimal degree of parallelism

For the third load test, this setting was changed from 0 to -1. The following table summarizes the results:

METRIC	TEST 1	TEST 2	TEST 3
Throughput (req/sec)	19	23	42
Average latency (ms)	669	569	215
Successful requests	9.8 K	11 K	20 K
Throttled requests	2.72 K	0	0

From the load test graph, not only is the overall throughput much higher (the orange line), but throughput also keeps pace with the load (the purple line).

We can verify that the Cosmos DB client is making queries in parallel by looking at the end-to-end transaction view:

Interestingly, a side effect of increasing the throughput is that the number of RUs consumed per second also increases. Although Cosmos DB did not throttle any requests during this test, the consumption was close to the provisioned RU limit:

This graph might be a signal to further scale out the database. However, it turns out that we can optimize the query instead.

Step 4: Optimize the query

The previous load test showed better performance in terms of latency and throughput. Average request latency was reduced by 68% and throughput increased 220%. However, the cross-partition query is a concern.

The problem with cross-partition queries is that you pay for RU across every partition. If the query is only run occasionally — say, once an hour — it might not matter. But whenever you see a read-heavy workload that involves a cross-partition query, you should see whether the query can be optimized by including a partition key. (You might need to redesign the collection to use a different partition key.)

Here's the query for this particular scenario:

```
SELECT * FROM c
WHERE c.ownerId = <ownerIdValue> and
      c.year = <yearValue> and
      c.month = <monthValue>
```

This query selects records that match a particular owner ID and month/year. In the original design, none of these properties is the partition key. That requires the client to fan out the query to each physical partition and gather the results. To improve query performance, the development team changed the design so that owner ID is the partition key for the collection. That way, the query can target a specific physical partition. (Cosmos DB handles this automatically; you don't have to manage the mapping between partition key values and physical partitions.)

After switching the collection to the new partition key, there was a dramatic improvement in RU consumption, which translates directly into lower costs.

METRIC	TEST 1	TEST 2	TEST 3	TEST 4
RUs per operation	29	29	29	3.4
Calls per operation	11	9	10	1

The end-to-end transaction view shows that as predicted, the query reads only one physical partition:

The load test shows improved throughput and latency:

METRIC	TEST 1	TEST 2	TEST 3	TEST 4
Throughput (req/sec)	19	23	42	59
Average latency (ms)	669	569	215	176
Successful requests	9.8 K	11 K	20 K	29 K
Throttled requests	2.72 K	0	0	0

A consequence of the improved performance is that node CPU utilization becomes very high:

Toward the end of the load test, average CPU reached about 90%, and maximum CPU reached 100%. This metric indicates that CPU is the next bottleneck in the system. If higher throughput is needed, the next step might be scaling out the Delivery service to more instances.

Summary

For this scenario, the following bottlenecks were identified:

- Cosmos DB throttling requests due to insufficient RUs provisioned.
- High latency caused by querying multiple database partitions in serial.
- Inefficient cross-partition query, because the query did not include the partition key.

In addition, CPU utilization was identified as a potential bottleneck at higher scale. To diagnose these issues, the development team looked at:

- Latency and throughput from the load test.
- Cosmos DB errors and RU consumption.
- The end-to-end transaction view in Application Insight.
- CPU and memory utilization in Azure Monitor for containers.

Next steps

Review performance antipatterns

Performance tuning scenario: Event streaming with Azure Functions

11/19/2021 • 7 minutes to read • Edit Online

This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that we did for a sample application.

This article is part of a series. Read the first part here.

Scenario: Process a stream of events using Azure Functions.

In this scenario, a fleet of drones sends position data in real time to Azure IoT Hub. A Functions app receives the events, transforms the data into GeoJSON format, and writes the transformed data to Cosmos DB. Cosmos DB has native support for geospatial data, and Cosmos DB collections can be indexed for efficient spatial queries. For example, a client application could query for all drones within 1 km of a given location, or find all drones within a certain area.

These processing requirements are simple enough that they don't require a full-fledged stream processing engine. In particular, the processing doesn't join streams, aggregate data, or process across time windows. Based on these requirements, Azure Functions is a good fit for processing the messages. Cosmos DB can also scale to support very high write throughput.

Monitoring throughput

This scenario presents an interesting performance challenge. The data rate *per device* is known, but the number of devices may fluctuate. For this business scenario, the latency requirements are not particularly stringent. The reported position of a drone only needs to be accurate within a minute. That said, the function app must keep up with the average ingestion rate over time.

IoT Hub stores messages in a log stream. Incoming messages are appended to the tail of the stream. A reader of the stream — in this case, the function app — controls its own rate of traversing the stream. This decoupling of the read and write paths makes IoT Hub very efficient, but also means that a slow reader can fall behind. To detect this condition, the development team added a custom metric to measure message lateness. This metric records the delta between when a message arrives at IoT Hub, and when the function receives the message for processing.

```
var ticksUTCNow = DateTimeOffset.UtcNow;

// Track whether messages are arriving at the function late.
DateTime? firstMsgEnqueuedTicksUtc = messages[0]?.EnqueuedTimeUtc;
if (firstMsgEnqueuedTicksUtc.HasValue)
{
    CustomTelemetry.TrackMetric(
                        context,
                        "IoTHubMessagesReceivedFreshnessMsec",
                        (ticksUTCNow - firstMsgEnqueuedTicksUtc.Value).TotalMilliseconds);
}
```

The `TrackMetric` method writes a custom metric to Application Insights. For information about using `TrackMetric` inside an Azure Function, see Custom telemetry in C# function.

If the function keeps up with the volume of messages, this metric should stay at a low steady state. Some latency is unavoidable, so the value will never be zero. But if the function falls behind, the delta between enqueued time and processing time will start to go up.

Test 1: Baseline

The first load test showed an immediate problem: The Function app consistently received HTTP 429 errors from Cosmos DB, indicating that Cosmos DB was throttling the write requests.

In response, the team scaled Cosmos DB by increasing the number RUs allocated for the collection, but the errors continued. This seemed strange, because their back-of-envelope calculation showed that Cosmos DB should have no problem keeping up with the volume of write requests.

Later that day, one of the developers sent the following email to the team:

> I looked at Cosmos DB for the warm path. There's one thing I don't understand. The partition key is deliveryId, however we don't send deliveryId to Cosmos DB. Am I missing something?

That was the clue. Looking at the partition heat map, it turned out that all of the documents were landing on the same partition.

What you want to see in the heat map is an even distribution across all of the partitions. In this case, because every document was getting written to the same partition, adding RUs didn't help. The problem turned out to be a bug in the code. Although the Cosmos DB collection had a partition key, the Azure Function didn't actually include the partition key in the document. For more information about the partition heat map, see Determine the throughput distribution across partitions.

Test 2: Fix partitioning issue

When the team deployed a code fix and re-ran the test, Cosmos DB stopped throttling. For a while, everything looked good. But at a certain load, telemetry showed that the function was writing fewer documents that it should. The following graph shows messages receives from IoT Hub versus documents written to Cosmos DB. The yellow line is number of messages received per batch, and the green is the number of documents written per batch. These should be proportional. Instead, the number of database write operations per batch drops significantly at about 07:30.

The next graph shows the latency between when a message arrives at IoT Hub from a device, and when the function app processes that message. You can see that at the same point in time, the lateness spikes dramatically, levels off, and the declines.

The reason the value peaks at 5 minutes and then drops to zero is because the function app discards messages that are more than 5 minutes late:

```
foreach (var message in messages)
{
    // Drop stale messages,
    if (message.EnqueuedTimeUtc < cutoffTime)
    {
        log.Info($"Dropping late message batch. Enqueued time = {message.EnqueuedTimeUtc}, Cutoff =
{cutoffTime}");
        droppedMessages++;
        continue;
    }
```

You can see this in the graph when the lateness metric drops back to zero. In the meantime, data has been lost, because the function was throwing away messages.

What was happening? For this particular load test, the Cosmos DB collection had RUs to spare, so the bottleneck was not at the database. Rather, the problem was in the message processing loop. Simply put, the function was not writing documents quickly enough to keep up with the incoming volume of messages. Over time, it fell further and further behind.

Test 3: Parallel writes

If the time to process a message is the bottleneck, one solution is to process more messages in parallel. In this scenario:

- Increase the number of IoT Hub partitions. Each IoT Hub partition gets assigned one function instance at a time, so we would expect throughput to scale linearly with the number of partitions.
- Parallelize the document writes within the function.

To explore the second option, the team modified the function to support parallel writes. The original version of the function used the Cosmos DB output binding. The optimized version calls the Cosmos DB client directly and performs the writes in parallel using Task.WhenAll:

```
private async Task<(long documentsUpserted,
                    long droppedMessages,
                    long cosmosDbTotalMilliseconds)>
                ProcessMessagesFromEventHub(
                    int taskCount,
                    int numberOfDocumentsToUpsertPerTask,
                    EventData[] messages,
                    TraceWriter log)
{
    DateTimeOffset cutoffTime = DateTimeOffset.UtcNow.AddMinutes(-5);

    var tasks = new List<Task>();

    for (var i = 0; i < taskCount; i++)
    {
        var docsToUpsert = messages
                            .Skip(i * numberOfDocumentsToUpsertPerTask)
                            .Take(numberOfDocumentsToUpsertPerTask);
        // client will attempt to create connections to the data
        // nodes on Cosmos db's clusters on a range of port numbers
        tasks.Add(UpsertDocuments(i, docsToUpsert, cutoffTime, log));
    }

    await Task.WhenAll(tasks);

    return (this.UpsertedDocuments,
            this.DroppedMessages,
            this.CosmosDbTotalMilliseconds);
}
```

Note that race conditions are possible with approach. Suppose that two messages from the same drone happen to arrive in the same batch of messages. By writing them in parallel, the earlier message could overwrite the later message. For this particular scenario, the application can tolerate losing an occasional message. Drones send new position data every 5 seconds, so the data in Cosmos DB is updated continually. In other scenarios, however, it may be important to process messages strictly in order.

After deploying this code change, the application was able to ingest more than 2500 requests/sec, using an IoT Hub with 32 partitions.

Client-side Consideration

Overall client experience may be diminished by aggressive parallelization on server side. Consider leveraging Azure Cosmos DB bulk executor library (not shown in this implementation) which significantly reduces the client-side compute resources needed to saturate the throughput allocated to a Cosmos DB container. A single threaded application that writes data using the bulk import API achieves nearly ten times greater write throughput when compared to a multi-threaded application that writes data in parallel while saturating the client machine's CPU.

Summary

For this scenario, the following bottlenecks were identified:

- Hot write partition, due to a missing partition key value in the documents being written.
- Writing documents in serial per IoT Hub partition.

To diagnose these issues, the development team relied on the following metrics:

- Throttled requests in Cosmos DB.
- Partition heat map — Maximum consumed RUs per partition.
- Messages received versus documents created.
- Message lateness.

Next steps

Review performance antipatterns

Performance testing and antipatterns for cloud applications

11/19/2021 • 2 minutes to read • Edit Online

Performance antipatterns, much like design patterns, are common defective processes and implementations within organizations. These are common practices that are likely to cause scalability problems when an application is under pressure. Awareness of these practices can help simplify communication of high-level concepts amongst software practitioners.

Here is a common scenario: An application behaves well during performance testing. It's released to production, and begins to handle real workloads. At that point, it starts to perform poorly—rejecting user requests, stalling, or throwing exceptions. The development team is then faced with two questions:

- Why didn't this behavior show up during testing?
- How do we fix it?

The answer to the first question is straightforward. It's difficult to simulate real users in a test environment, along with their behavior patterns and the volumes of work they might perform. The only completely sure way to understand how a system behaves under load is to observe it in production. To be clear, we aren't suggesting that you should skip performance testing. Performance testing is crucial for getting baseline performance metrics. But you must be prepared to observe and correct performance issues when they arise in the live system.

The answer to the second question, how to fix the problem, is less straightforward. Any number of factors might contribute, and sometimes the problem only manifests under certain circumstances. Instrumentation and logging are key to finding the root cause, but you also have to know what to look for.

Based on our engagements with Microsoft Azure customers, we've identified some of the most common performance issues that customers see in production. For each antipattern, we describe why the antipattern typically occurs, symptoms of the antipattern, and techniques for resolving the problem. We also provide sample code that illustrates both the antipattern and a suggested scalability solution.

Some of these antipatterns may seem obvious when you read the descriptions, but they occur more often than you might think. Sometimes an application inherits a design that worked on-premises, but doesn't scale in the cloud. Or an application might start with a very clean design, but as new features are added, one or more of these antipatterns creeps in. Regardless, this guide will help you to identify and fix these antipatterns.

Catalog of antipatterns

Here is the list of the antipatterns that we've identified:

ANTIPATTERN	DESCRIPTION
Busy Database	Offloading too much processing to a data store.
Busy Front End	Moving resource-intensive tasks onto background threads.
Chatty I/O	Continually sending many small network requests.

ANTIPATTERN	DESCRIPTION
Extraneous Fetching	Retrieving more data than is needed, resulting in unnecessary I/O.
Improper Instantiation	Repeatedly creating and destroying objects that are designed to be shared and reused.
Monolithic Persistence	Using the same data store for data with very different usage patterns.
No Caching	Failing to cache data.
Noisy Neighbor	A single tenant uses a disproportionate amount of the resources.
Retry Storm	Retrying failed requests to a server too often.
Synchronous I/O	Blocking the calling thread while I/O completes.

Next steps

For more about performance tuning, see Performance tuning a distributed application

Busy Database antipattern

11/19/2021 • 7 minutes to read • Edit Online

Offloading processing to a database server can cause it to spend a significant proportion of time running code, rather than responding to requests to store and retrieve data.

Problem description

Many database systems can run code. Examples include stored procedures and triggers. Often, it's more efficient to perform this processing close to the data, rather than transmitting the data to a client application for processing. However, overusing these features can hurt performance, for several reasons:

- The database server may spend too much time processing, rather than accepting new client requests and fetching data.
- A database is usually a shared resource, so it can become a bottleneck during periods of high use.
- Runtime costs may be excessive if the data store is metered. That's particularly true of managed database services. For example, Azure SQL Database charges for Database Transaction Units (DTUs).
- Databases have finite capacity to scale up, and it's not trivial to scale a database horizontally. Therefore, it may be better to move processing into a compute resource, such as a VM or App Service app, that can easily scale out.

This antipattern typically occurs because:

- The database is viewed as a service rather than a repository. An application might use the database server to format data (for example, converting to XML), manipulate string data, or perform complex calculations.
- Developers try to write queries whose results can be displayed directly to users. For example, a query might combine fields or format dates, times, and currency according to locale.
- Developers are trying to correct the Extraneous Fetching antipattern by pushing computations to the database.
- Stored procedures are used to encapsulate business logic, perhaps because they are considered easier to maintain and update.

The following example retrieves the 20 most valuable orders for a specified sales territory and formats the results as XML. It uses Transact-SQL functions to parse the data and convert the results to XML. You can find the complete sample here.

```
SELECT TOP 20
  soh.[SalesOrderNumber]  AS '@OrderNumber',
  soh.[Status]            AS '@Status',
  soh.[ShipDate]          AS '@ShipDate',
  YEAR(soh.[OrderDate])   AS '@OrderDateYear',
  MONTH(soh.[OrderDate])  AS '@OrderDateMonth',
  soh.[DueDate]           AS '@DueDate',
  FORMAT(ROUND(soh.[SubTotal],2),'C')
                          AS '@SubTotal',
  FORMAT(ROUND(soh.[TaxAmt],2),'C')
                          AS '@TaxAmt',
  FORMAT(ROUND(soh.[TotalDue],2),'C')
                          AS '@TotalDue',
  CASE WHEN soh.[TotalDue] > 5000 THEN 'Y' ELSE 'N' END
                          AS '@ReviewRequired',
  (
  SELECT
    c.[AccountNumber]     AS '@AccountNumber',
    UPPER(LTRIM(RTRIM(REPLACE(
    CONCAT( p.[Title], ' ', p.[FirstName], ' ', p.[MiddleName], ' ', p.[LastName], ' ', p.[Suffix]),
    '  ', ' '))))         AS '@FullName'
  FROM [Sales].[Customer] c
    INNER JOIN [Person].[Person] p
  ON c.[PersonID] = p.[BusinessEntityID]
  WHERE c.[CustomerID] = soh.[CustomerID]
  FOR XML PATH ('Customer'), TYPE
  ),

  (
  SELECT
    sod.[OrderQty]      AS '@Quantity',
    FORMAT(sod.[UnitPrice],'C')
                        AS '@UnitPrice',
    FORMAT(ROUND(sod.[LineTotal],2),'C')
                        AS '@LineTotal',
    sod.[ProductID]     AS '@ProductId',
    CASE WHEN (sod.[ProductID] >= 710) AND (sod.[ProductID] <= 720) AND (sod.[OrderQty] >= 5) THEN 'Y' ELSE
'N' END
                        AS '@InventoryCheckRequired'

  FROM [Sales].[SalesOrderDetail] sod
  WHERE sod.[SalesOrderID] = soh.[SalesOrderID]
  ORDER BY sod.[SalesOrderDetailID]
  FOR XML PATH ('LineItem'), TYPE, ROOT('OrderLineItems')
  )

FROM [Sales].[SalesOrderHeader] soh
WHERE soh.[TerritoryId] = @TerritoryId
ORDER BY soh.[TotalDue] DESC
FOR XML PATH ('Order'), ROOT('Orders')
```

Clearly, this is complex query. As we'll see later, it turns out to use significant processing resources on the database server.

How to fix the problem

Move processing from the database server into other application tiers. Ideally, you should limit the database to performing data access operations, using only the capabilities that the database is optimized for, such as aggregation in an RDBMS.

For example, the previous Transact-SQL code can be replaced with a statement that simply retrieves the data to be processed.

```
SELECT
soh.[SalesOrderNumber]  AS [OrderNumber],
soh.[Status]            AS [Status],
soh.[OrderDate]         AS [OrderDate],
soh.[DueDate]           AS [DueDate],
soh.[ShipDate]          AS [ShipDate],
soh.[SubTotal]          AS [SubTotal],
soh.[TaxAmt]            AS [TaxAmt],
soh.[TotalDue]          AS [TotalDue],
c.[AccountNumber]       AS [AccountNumber],
p.[Title]               AS [CustomerTitle],
p.[FirstName]           AS [CustomerFirstName],
p.[MiddleName]          AS [CustomerMiddleName],
p.[LastName]            AS [CustomerLastName],
p.[Suffix]              AS [CustomerSuffix],
sod.[OrderQty]          AS [Quantity],
sod.[UnitPrice]         AS [UnitPrice],
sod.[LineTotal]         AS [LineTotal],
sod.[ProductID]         AS [ProductId]
FROM [Sales].[SalesOrderHeader] soh
INNER JOIN [Sales].[Customer] c ON soh.[CustomerID] = c.[CustomerID]
INNER JOIN [Person].[Person] p ON c.[PersonID] = p.[BusinessEntityID]
INNER JOIN [Sales].[SalesOrderDetail] sod ON soh.[SalesOrderID] = sod.[SalesOrderID]
WHERE soh.[TerritoryId] = @TerritoryId
AND soh.[SalesOrderId] IN (
    SELECT TOP 20 SalesOrderId
    FROM [Sales].[SalesOrderHeader] soh
    WHERE soh.[TerritoryId] = @TerritoryId
    ORDER BY soh.[TotalDue] DESC)
ORDER BY soh.[TotalDue] DESC, sod.[SalesOrderDetailID]
```

The application then uses the .NET Framework `System.Xml.Linq` APIs to format the results as XML.

```
// Create a new SqlCommand to run the Transact-SQL query
using (var command = new SqlCommand(...))
{
    command.Parameters.AddWithValue("@TerritoryId", id);

    // Run the query and create the initial XML document
    using (var reader = await command.ExecuteReaderAsync())
    {
        var lastOrderNumber = string.Empty;
        var doc = new XDocument();
        var orders = new XElement("Orders");
        doc.Add(orders);

        XElement lineItems = null;
        // Fetch each row in turn, format the results as XML, and add them to the XML document
        while (await reader.ReadAsync())
        {
            var orderNumber = reader["OrderNumber"].ToString();
            if (orderNumber != lastOrderNumber)
            {
                lastOrderNumber = orderNumber;

                var order = new XElement("Order");
                orders.Add(order);
                var customer = new XElement("Customer");
                lineItems = new XElement("OrderLineItems");
                order.Add(customer, lineItems);

                var orderDate = (DateTime)reader["OrderDate"];
                var totalDue = (Decimal)reader["TotalDue"];
                var reviewRequired = totalDue > 5000 ? 'Y' : 'N';

                order.Add(
```

```
                    new XAttribute("OrderNumber", orderNumber),
                    new XAttribute("Status", reader["Status"]),
                    new XAttribute("ShipDate", reader["ShipDate"]),
                    ... // More attributes, not shown.

                    var fullName = string.Join(" ",
                        reader["CustomerTitle"],
                        reader["CustomerFirstName"],
                        reader["CustomerMiddleName"],
                        reader["CustomerLastName"],
                        reader["CustomerSuffix"]
                    )
                   .Replace("  ", " ") //remove double spaces
                   .Trim()
                   .ToUpper();

                customer.Add(
                    new XAttribute("AccountNumber", reader["AccountNumber"]),
                    new XAttribute("FullName", fullName));
            }

            var productId = (int)reader["ProductID"];
            var quantity = (short)reader["Quantity"];
            var inventoryCheckRequired = (productId >= 710 && productId <= 720 && quantity >= 5) ? 'Y' :
'N';

            lineItems.Add(
                new XElement("LineItem",
                    new XAttribute("Quantity", quantity),
                    new XAttribute("UnitPrice", ((Decimal)reader["UnitPrice"]).ToString("C")),
                    new XAttribute("LineTotal", RoundAndFormat(reader["LineTotal"])),
                    new XAttribute("ProductId", productId),
                    new XAttribute("InventoryCheckRequired", inventoryCheckRequired)
                ));
        }
        // Match the exact formatting of the XML returned from SQL
        var xml = doc
            .ToString(SaveOptions.DisableFormatting)
            .Replace(" />", "/>");
    }
}
```

NOTE

This code is somewhat complex. For a new application, you might prefer to use a serialization library. However, the assumption here is that the development team is refactoring an existing application, so the method needs to return the exact same format as the original code.

Considerations

- Many database systems are highly optimized to perform certain types of data processing, such as calculating aggregate values over large datasets. Don't move those types of processing out of the database.
- Do not relocate processing if doing so causes the database to transfer far more data over the network. See the Extraneous Fetching antipattern.
- If you move processing to an application tier, that tier may need to scale out to handle the additional work.

How to detect the problem

Symptoms of a busy database include a disproportionate decline in throughput and response times in operations that access the database.

You can perform the following steps to help identify this problem:

1. Use performance monitoring to identify how much time the production system spends performing database activity.
2. Examine the work performed by the database during these periods.
3. If you suspect that particular operations might cause too much database activity, perform load testing in a controlled environment. Each test should run a mixture of the suspect operations with a variable user load. Examine the telemetry from the load tests to observe how the database is used.
4. If the database activity reveals significant processing but little data traffic, review the source code to determine whether the processing can better be performed elsewhere.

If the volume of database activity is low or response times are relatively fast, then a busy database is unlikely to be a performance problem.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Monitor the volume of database activity

The following graph shows the results of running a load test against the sample application, using a step load of up to 50 concurrent users. The volume of requests quickly reaches a limit and stays at that level, while the average response time steadily increases. A logarithmic scale is used for those two metrics.

This line graph shows user load, requests per second, and average response time. The graph shows that response time increases as load increases.

The next graph shows CPU utilization and DTUs as a percentage of service quota. DTUs provide a measure of how much processing the database performs. The graph shows that CPU and DTU utilization both quickly

reached 100%.

This line graph shows CPU percentage and DTU percentage over time. The graph shows that both quickly reach 100%.

Examine the work performed by the database

It could be that the tasks performed by the database are genuine data access operations, rather than processing, so it is important to understand the SQL statements being run while the database is busy. Monitor the system to capture the SQL traffic and correlate the SQL operations with application requests.

If the database operations are purely data access operations, without a lot of processing, then the problem might be Extraneous Fetching.

Implement the solution and verify the result

The following graph shows a load test using the updated code. Throughput is significantly higher, over 400 requests per second versus 12 earlier. The average response time is also much lower, just above 0.1 seconds compared to over 4 seconds.

This line graph shows user load, requests per second, and average response time. The graph shows that response time remains roughly constant throughout the load test.

CPU and DTU utilization shows that the system took longer to reach saturation, despite the increased throughput.

This line graph shows CPU percentage and DTU percentage over time. The graph shows that CPU and DTU take longer to reach 100% than previously.

Related resources

- Extraneous Fetching antipattern

Busy Front End antipattern

11/19/2021 • 8 minutes to read • Edit Online

Performing asynchronous work on a large number of background threads can starve other concurrent foreground tasks of resources, decreasing response times to unacceptable levels.

Problem description

Resource-intensive tasks can increase the response times for user requests and cause high latency. One way to improve response times is to offload a resource-intensive task to a separate thread. This approach lets the application stay responsive while processing happens in the background. However, tasks that run on a background thread still consume resources. If there are too many of them, they can starve the threads that are handling requests.

> **NOTE**
>
> The term *resource* can encompass many things, such as CPU utilization, memory occupancy, and network or disk I/O.

This problem typically occurs when an application is developed as monolithic piece of code, with all of the business logic combined into a single tier shared with the presentation layer.

Here's an example using ASP.NET that demonstrates the problem. You can find the complete sample here.

```
public class WorkInFrontEndController : ApiController
{
    [HttpPost]
    [Route("api/workinfrontend")]
    public HttpResponseMessage Post()
    {
        new Thread(() =>
        {
            //Simulate processing
            Thread.SpinWait(Int32.MaxValue / 100);
        }).Start();

        return Request.CreateResponse(HttpStatusCode.Accepted);
    }
}

public class UserProfileController : ApiController
{
    [HttpGet]
    [Route("api/userprofile/{id}")]
    public UserProfile Get(int id)
    {
        //Simulate processing
        return new UserProfile() { FirstName = "Alton", LastName = "Hudgens" };
    }
}
```

- The `Post` method in the `WorkInFrontEnd` controller implements an HTTP POST operation. This operation simulates a long-running, CPU-intensive task. The work is performed on a separate thread, in an attempt to enable the POST operation to complete quickly.
- The `Get` method in the `UserProfile` controller implements an HTTP GET operation. This method is much

less CPU intensive.

The primary concern is the resource requirements of the `Post` method. Although it puts the work onto a background thread, the work can still consume considerable CPU resources. These resources are shared with other operations being performed by other concurrent users. If a moderate number of users send this request at the same time, overall performance is likely to suffer, slowing down all operations. Users might experience significant latency in the `Get` method, for example.

How to fix the problem

Move processes that consume significant resources to a separate back end.

With this approach, the front end puts resource-intensive tasks onto a message queue. The back end picks up the tasks for asynchronous processing. The queue also acts as a load leveler, buffering requests for the back end. If the queue length becomes too long, you can configure autoscaling to scale out the back end.

Here is a revised version of the previous code. In this version, the `Post` method puts a message on a Service Bus queue.

```
public class WorkInBackgroundController : ApiController
{
    private static readonly QueueClient QueueClient;
    private static readonly string QueueName;
    private static readonly ServiceBusQueueHandler ServiceBusQueueHandler;

    public WorkInBackgroundController()
    {
        var serviceBusConnectionString = ...;
        QueueName = ...;
        ServiceBusQueueHandler = new ServiceBusQueueHandler(serviceBusConnectionString);
        QueueClient = ServiceBusQueueHandler.GetQueueClientAsync(QueueName).Result;
    }

    [HttpPost]
    [Route("api/workinbackground")]
    public async Task<long> Post()
    {
        return await ServiceBusQueueHandler.AddWorkLoadToQueueAsync(QueueClient, QueueName, 0);
    }
}
```

The back end pulls messages from the Service Bus queue and does the processing.

```
public async Task RunAsync(CancellationToken cancellationToken)
{
    this._queueClient.OnMessageAsync(
        // This lambda is invoked for each message received.
        async (receivedMessage) =>
        {
            try
            {
                // Simulate processing of message
                Thread.SpinWait(Int32.MaxValue / 1000);

                await receivedMessage.CompleteAsync();
            }
            catch
            {
                receivedMessage.Abandon();
            }
        });
}
```

Considerations

- This approach adds some additional complexity to the application. You must handle queuing and dequeuing safely to avoid losing requests in the event of a failure.
- The application takes a dependency on an additional service for the message queue.
- The processing environment must be sufficiently scalable to handle the expected workload and meet the required throughput targets.
- While this approach should improve overall responsiveness, the tasks that are moved to the back end may take longer to complete.

How to detect the problem

Symptoms of a busy front end include high latency when resource-intensive tasks are being performed. End users are likely to report extended response times or failures caused by services timing out. These failures could also return HTTP 500 (Internal Server) errors or HTTP 503 (Service Unavailable) errors. Examine the event logs for the web server, which are likely to contain more detailed information about the causes and circumstances of the errors.

You can perform the following steps to help identify this problem:

1. Perform process monitoring of the production system, to identify points when response times slow down.
2. Examine the telemetry data captured at these points to determine the mix of operations being performed and the resources being used.
3. Find any correlations between poor response times and the volumes and combinations of operations that were happening at those times.
4. Load test each suspected operation to identify which operations are consuming resources and starving other operations.
5. Review the source code for those operations to determine why they might cause excessive resource consumption.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Identify points of slowdown

Instrument each method to track the duration and resources consumed by each request. Then monitor the

application in production. This can provide an overall view of how requests compete with each other. During periods of stress, slow-running resource-hungry requests will likely affect other operations, and this behavior can be observed by monitoring the system and noting the drop off in performance.

The following image shows a monitoring dashboard. (We used AppDynamics for our tests.) Initially, the system has light load. Then users start requesting the `UserProfile` GET method. The performance is reasonably good until other users start issuing requests to the `WorkInFrontEnd` POST method. At that point, response times increase dramatically (first arrow). Response times only improve after the volume of requests to the `WorkInFrontEnd` controller diminishes (second arrow).

Examine telemetry data and find correlations

The next image shows some of the metrics gathered to monitor resource utilization during the same interval. At first, few users are accessing the system. As more users connect, CPU utilization becomes very high (100%). Also notice that the network I/O rate initially goes up as CPU usage rises. But once CPU usage peaks, network I/O actually goes down. That's because the system can only handle a relatively small number of requests once the CPU is at capacity. As users disconnect, the CPU load tails off.

At this point, it appears the `Post` method in the `WorkInFrontEnd` controller is a prime candidate for closer examination. Further work in a controlled environment is needed to confirm the hypothesis.

Perform load testing

The next step is to perform tests in a controlled environment. For example, run a series of load tests that include and then omit each request in turn to see the effects.

The graph below shows the results of a load test performed against an identical deployment of the cloud service used in the previous tests. The test used a constant load of 500 users performing the `Get` operation in the `UserProfile` controller, along with a step load of users performing the `Post` operation in the `WorkInFrontEnd` controller.

Initially, the step load is 0, so the only active users are performing the `UserProfile` requests. The system is able to respond to approximately 500 requests per second. After 60 seconds, a load of 100 additional users starts sending POST requests to the `WorkInFrontEnd` controller. Almost immediately, the workload sent to the `UserProfile` controller drops to about 150 requests per second. This is due to the way the load-test runner functions. It waits for a response before sending the next request, so the longer it takes to receive a response, the lower the request rate.

As more users send POST requests to the `WorkInFrontEnd` controller, the response rate of the `UserProfile` controller continues to drop. But note that the volume of requests handled by the `WorkInFrontEnd` controller remains relatively constant. The saturation of the system becomes apparent as the overall rate of both requests tends toward a steady but low limit.

Review the source code

The final step is to look at the source code. The development team was aware that the `Post` method could take a considerable amount of time, which is why the original implementation used a separate thread. That solved the immediate problem, because the `Post` method did not block waiting for a long-running task to complete.

However, the work performed by this method still consumes CPU, memory, and other resources. Enabling this process to run asynchronously might actually damage performance, as users can trigger a large number of these operations simultaneously, in an uncontrolled manner. There is a limit to the number of threads that a server can run. Past this limit, the application is likely to get an exception when it tries to start a new thread.

> **NOTE**
>
> This doesn't mean you should avoid asynchronous operations. Performing an asynchronous await on a network call is a recommended practice. (See the Synchronous I/O antipattern.) The problem here is that CPU-intensive work was spawned on another thread.

Implement the solution and verify the result

The following image shows performance monitoring after the solution was implemented. The load was similar to that shown earlier, but the response times for the `UserProfile` controller are now much faster. The volume of requests increased over the same duration, from 2,759 to 23,565.

Note that the `WorkInBackground` controller also handled a much larger volume of requests. However, you can't make a direct comparison in this case, because the work being performed in this controller is very different from the original code. The new version simply queues a request, rather than performing a time consuming calculation. The main point is that this method no longer drags down the entire system under load.

CPU and network utilization also show the improved performance. The CPU utilization never reached 100%, and the volume of handled network requests was far greater than earlier, and did not tail off until the workload dropped.

The following graph shows the results of a load test. The overall volume of requests serviced is greatly improved compared to the earlier tests.

Related guidance

- Autoscaling best practices
- Background jobs best practices
- Queue-Based Load Leveling pattern
- Web Queue Worker architecture style

Chatty I/O antipattern

11/19/2021 • 9 minutes to read • Edit Online

The cumulative effect of a large number of I/O requests can have a significant impact on performance and responsiveness.

Problem description

Network calls and other I/O operations are inherently slow compared to compute tasks. Each I/O request typically has significant overhead, and the cumulative effect of numerous I/O operations can slow down the system. Here are some common causes of chatty I/O.

Reading and writing individual records to a database as distinct requests

The following example reads from a database of products. There are three tables, `Product`, `ProductSubcategory`, and `ProductPriceListHistory`. The code retrieves all of the products in a subcategory, along with the pricing information, by executing a series of queries:

1. Query the subcategory from the `ProductSubcategory` table.
2. Find all products in that subcategory by querying the `Product` table.
3. For each product, query the pricing data from the `ProductPriceListHistory` table.

The application uses Entity Framework to query the database. You can find the complete sample here.

```
public async Task<IHttpActionResult> GetProductsInSubCategoryAsync(int subcategoryId)
{
    using (var context = GetContext())
    {
        // Get product subcategory.
        var productSubcategory = await context.ProductSubcategories
                .Where(psc => psc.ProductSubcategoryId == subcategoryId)
                .FirstOrDefaultAsync();

        // Find products in that category.
        productSubcategory.Product = await context.Products
            .Where(p => subcategoryId == p.ProductSubcategoryId)
            .ToListAsync();

        // Find price history for each product.
        foreach (var prod in productSubcategory.Product)
        {
            int productId = prod.ProductId;
            var productListPriceHistory = await context.ProductListPriceHistory
                .Where(pl => pl.ProductId == productId)
                .ToListAsync();
            prod.ProductListPriceHistory = productListPriceHistory;
        }
        return Ok(productSubcategory);
    }
}
```

This example shows the problem explicitly, but sometimes an O/RM can mask the problem, if it implicitly fetches child records one at a time. This is known as the "N+1 problem".

Implementing a single logical operation as a series of HTTP requests

This often happens when developers try to follow an object-oriented paradigm, and treat remote objects as if

they were local objects in memory. This can result in too many network round trips. For example, the following web API exposes the individual properties of `User` objects through individual HTTP GET methods.

```
public class UserController : ApiController
{
    [HttpGet]
    [Route("users/{id:int}/username")]
    public HttpResponseMessage GetUserName(int id)
    {
        ...
    }

    [HttpGet]
    [Route("users/{id:int}/gender")]
    public HttpResponseMessage GetGender(int id)
    {
        ...
    }

    [HttpGet]
    [Route("users/{id:int}/dateofbirth")]
    public HttpResponseMessage GetDateOfBirth(int id)
    {
        ...
    }
}
```

While there's nothing technically wrong with this approach, most clients will probably need to get several properties for each `User`, resulting in client code like the following.

```
HttpResponseMessage response = await client.GetAsync("users/1/username");
response.EnsureSuccessStatusCode();
var userName = await response.Content.ReadAsStringAsync();

response = await client.GetAsync("users/1/gender");
response.EnsureSuccessStatusCode();
var gender = await response.Content.ReadAsStringAsync();

response = await client.GetAsync("users/1/dateofbirth");
response.EnsureSuccessStatusCode();
var dob = await response.Content.ReadAsStringAsync();
```

Reading and writing to a file on disk

File I/O involves opening a file and moving to the appropriate point before reading or writing data. When the operation is complete, the file might be closed to save operating system resources. An application that continually reads and writes small amounts of information to a file will generate significant I/O overhead. Small write requests can also lead to file fragmentation, slowing subsequent I/O operations still further.

The following example uses a `FileStream` to write a `Customer` object to a file. Creating the `FileStream` opens the file, and disposing it closes the file. (The `using` statement automatically disposes the `FileStream` object.) If the application calls this method repeatedly as new customers are added, the I/O overhead can accumulate quickly.

```
private async Task SaveCustomerToFileAsync(Customer customer)
{
    using (Stream fileStream = new FileStream(CustomersFileName, FileMode.Append))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        byte [] data = null;
        using (MemoryStream memStream = new MemoryStream())
        {
            formatter.Serialize(memStream, customer);
            data = memStream.ToArray();
        }
        await fileStream.WriteAsync(data, 0, data.Length);
    }
}
```

How to fix the problem

Reduce the number of I/O requests by packaging the data into larger, fewer requests.

Fetch data from a database as a single query, instead of several smaller queries. Here's a revised version of the code that retrieves product information.

```
public async Task<IHttpActionResult> GetProductCategoryDetailsAsync(int subCategoryId)
{
    using (var context = GetContext())
    {
        var subCategory = await context.ProductSubcategories
                .Where(psc => psc.ProductSubcategoryId == subCategoryId)
                .Include("Product.ProductListPriceHistory")
                .FirstOrDefaultAsync();

        if (subCategory == null)
            return NotFound();

        return Ok(subCategory);
    }
}
```

Follow REST design principles for web APIs. Here's a revised version of the web API from the earlier example. Instead of separate GET methods for each property, there is a single GET method that returns the `User`. This results in a larger response body per request, but each client is likely to make fewer API calls.

```
public class UserController : ApiController
{
    [HttpGet]
    [Route("users/{id:int}")]
    public HttpResponseMessage GetUser(int id)
    {
        ...
    }
}

// Client code
HttpResponseMessage response = await client.GetAsync("users/1");
response.EnsureSuccessStatusCode();
var user = await response.Content.ReadAsStringAsync();
```

For file I/O, consider buffering data in memory and then writing the buffered data to a file as a single operation. This approach reduces the overhead from repeatedly opening and closing the file, and helps to reduce fragmentation of the file on disk.

```
// Save a list of customer objects to a file
private async Task SaveCustomerListToFileAsync(List<Customer> customers)
{
    using (Stream fileStream = new FileStream(CustomersFileName, FileMode.Append))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        foreach (var customer in customers)
        {
            byte[] data = null;
            using (MemoryStream memStream = new MemoryStream())
            {
                formatter.Serialize(memStream, customer);
                data = memStream.ToArray();
            }
            await fileStream.WriteAsync(data, 0, data.Length);
        }
    }
}

// In-memory buffer for customers.
List<Customer> customers = new List<Customers>();

// Create a new customer and add it to the buffer
var customer = new Customer(...);
customers.Add(customer);

// Add more customers to the list as they are created
...

// Save the contents of the list, writing all customers in a single operation
await SaveCustomerListToFileAsync(customers);
```

Considerations

- The first two examples make *fewer* I/O calls, but each one retrieves *more* information. You must consider the tradeoff between these two factors. The right answer will depend on the actual usage patterns. For example, in the web API example, it might turn out that clients often need just the user name. In that case, it might make sense to expose it as a separate API call. For more information, see the Extraneous Fetching antipattern.
- When reading data, do not make your I/O requests too large. An application should only retrieve the information that it is likely to use.
- Sometimes it helps to partition the information for an object into two chunks, *frequently accessed data* that accounts for most requests, and *less frequently accessed data* that is used rarely. Often the most frequently accessed data is a relatively small portion of the total data for an object, so returning just that portion can save significant I/O overhead.
- When writing data, avoid locking resources for longer than necessary, to reduce the chances of contention during a lengthy operation. If a write operation spans multiple data stores, files, or services, then adopt an eventually consistent approach. See Data Consistency guidance.
- If you buffer data in memory before writing it, the data is vulnerable if the process crashes. If the data rate typically has bursts or is relatively sparse, it may be safer to buffer the data in an external durable queue such as Event Hubs.
- Consider caching data that you retrieve from a service or a database. This can help to reduce the volume of I/O by avoiding repeated requests for the same data. For more information, see Caching best practices.

How to detect the problem

Symptoms of chatty I/O include high latency and low throughput. End users are likely to report extended response times or failures caused by services timing out, due to increased contention for I/O resources.

You can perform the following steps to help identify the causes of any problems:

1. Perform process monitoring of the production system to identify operations with poor response times.
2. Perform load testing of each operation identified in the previous step.
3. During the load tests, gather telemetry data about the data access requests made by each operation.
4. Gather detailed statistics for each request sent to a data store.
5. Profile the application in the test environment to establish where possible I/O bottlenecks might be occurring.

Look for any of these symptoms:

- A large number of small I/O requests made to the same file.
- A large number of small network requests made by an application instance to the same service.
- A large number of small requests made by an application instance to the same data store.
- Applications and services becoming I/O bound.

Example diagnosis

The following sections apply these steps to the example shown earlier that queries a database.

Load test the application

This graph shows the results of load testing. Median response time is measured in tens of seconds per request. The graph shows very high latency. With a load of 1000 users, a user might have to wait for nearly a minute to see the results of a query.

> **NOTE**
> The application was deployed as an Azure App Service web app, using Azure SQL Database. The load test used a simulated step workload of up to 1000 concurrent users. The database was configured with a connection pool supporting up to 1000 concurrent connections, to reduce the chance that contention for connections would affect the results.

Monitor the application

You can use an application performance monitoring (APM) package to capture and analyze the key metrics that might identify chatty I/O. Which metrics are important will depend on the I/O workload. For this example, the interesting I/O requests were the database queries.

The following image shows results generated using New Relic APM. The average database response time peaked at approximately 5.6 seconds per request during the maximum workload. The system was able to support an average of 410 requests per minute throughout the test.

Gather detailed data access information

Digging deeper into the monitoring data shows the application executes three different SQL SELECT statements. These correspond to the requests generated by Entity Framework to fetch data from the `ProductListPriceHistory`, `Product`, and `ProductSubcategory` tables. Furthermore, the query that retrieves data from the `ProductListPriceHistory` table is by far the most frequently executed SELECT statement, by an order of magnitude.

It turns out that the `GetProductsInSubCategoryAsync` method, shown earlier, performs 45 SELECT queries. Each query causes the application to open a new SQL connection.

NOTE

This image shows trace information for the slowest instance of the `GetProductsInSubCategoryAsync` operation in the load test. In a production environment, it's useful to examine traces of the slowest instances, to see if there is a pattern that suggests a problem. If you just look at the average values, you might overlook problems that will get dramatically worse under load.

The next image shows the actual SQL statements that were issued. The query that fetches price information is run for each individual product in the product subcategory. Using a join would considerably reduce the number of database calls.

If you are using an O/RM, such as Entity Framework, tracing the SQL queries can provide insight into how the O/RM translates programmatic calls into SQL statements, and indicate areas where data access might be optimized.

Implement the solution and verify the result

Rewriting the call to Entity Framework produced the following results.

This load test was performed on the same deployment, using the same load profile. This time the graph shows much lower latency. The average request time at 1000 users is between 5 and 6 seconds, down from nearly a minute.

This time the system supported an average of 3,970 requests per minute, compared to 410 for the earlier test.

Tracing the SQL statement shows that all the data is fetched in a single SELECT statement. Although this query is considerably more complex, it is performed only once per operation. And while complex joins can become expensive, relational database systems are optimized for this type of query.

Related resources

- API Design best practices
- Caching best practices
- Data Consistency Primer
- Extraneous Fetching antipattern
- No Caching antipattern

Extraneous Fetching antipattern

11/19/2021 • 10 minutes to read • Edit Online

Anti-patterns are common design flaws that can break your software or applications under stress situations and should not be overlooked. In an *extraneous fetching antipattern*, more than needed data is retrieved for a business operation, often resulting in unnecessary I/O overhead and reduced responsiveness.

Examples of extraneous fetching antipattern

This antipattern can occur if the application tries to minimize I/O requests by retrieving all of the data that it *might* need. This is often a result of overcompensating for the Chatty I/O antipattern. For example, an application might fetch the details for every product in a database. But the user may need just a subset of the details (some may not be relevant to customers), and probably doesn't need to see *all* of the products at once. Even if the user is browsing the entire catalog, it would make sense to paginate the results—showing 20 at a time, for example.

Another source of this problem is following poor programming or design practices. For example, the following code uses Entity Framework to fetch the complete details for every product. Then it filters the results to return only a subset of the fields, discarding the rest. You can find the complete sample here.

```
public async Task<IHttpActionResult> GetAllFieldsAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Execute the query. This happens at the database.
        var products = await context.Products.ToListAsync();

        // Project fields from the query results. This happens in application memory.
        var result = products.Select(p => new ProductInfo { Id = p.ProductId, Name = p.Name });
        return Ok(result);
    }
}
```

In the next example, the application retrieves data to perform an aggregation that could be done by the database instead. The application calculates total sales by getting every record for all orders sold, and then computing the sum over those records. You can find the complete sample here.

```
public async Task<IHttpActionResult> AggregateOnClientAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Fetch all order totals from the database.
        var orderAmounts = await context.SalesOrderHeaders.Select(soh => soh.TotalDue).ToListAsync();

        // Sum the order totals in memory.
        var total = orderAmounts.Sum();
        return Ok(total);
    }
}
```

The next example shows a subtle problem caused by the way Entity Framework uses LINQ to Entities.

```
var query = from p in context.Products.AsEnumerable()
            where p.SellStartDate < DateTime.Now.AddDays(-7) // AddDays cannot be mapped by LINQ to Entities
            select ...;

List<Product> products = query.ToList();
```

The application is trying to find products with a `SellStartDate` more than a week old. In most cases, LINQ to Entities would translate a `where` clause to a SQL statement that is executed by the database. In this case, however, LINQ to Entities cannot map the `AddDays` method to SQL. Instead, every row from the `Product` table is returned, and the results are filtered in memory.

The call to `AsEnumerable` is a hint that there is a problem. This method converts the results to an `IEnumerable` interface. Although `IEnumerable` supports filtering, the filtering is done on the *client* side, not the database. By default, LINQ to Entities uses `IQueryable`, which passes the responsibility for filtering to the data source.

How to fix extraneous fetching antipattern

Avoid fetching large volumes of data that may quickly become outdated or might be discarded, and only fetch the data needed for the operation being performed.

Instead of getting every column from a table and then filtering them, select the columns that you need from the database.

```
public async Task<IHttpActionResult> GetRequiredFieldsAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Project fields as part of the query itself
        var result = await context.Products
            .Select(p => new ProductInfo {Id = p.ProductId, Name = p.Name})
            .ToListAsync();
        return Ok(result);
    }
}
```

Similarly, perform aggregation in the database and not in application memory.

```
public async Task<IHttpActionResult> AggregateOnDatabaseAsync()
{
    using (var context = new AdventureWorksContext())
    {
        // Sum the order totals as part of the database query.
        var total = await context.SalesOrderHeaders.SumAsync(soh => soh.TotalDue);
        return Ok(total);
    }
}
```

When using Entity Framework, ensure that LINQ queries are resolved using the `IQueryable` interface and not `IEnumerable`. You may need to adjust the query to use only functions that can be mapped to the data source. The earlier example can be refactored to remove the `AddDays` method from the query, allowing filtering to be done by the database.

```
DateTime dateSince = DateTime.Now.AddDays(-7); // AddDays has been factored out.
var query = from p in context.Products
            where p.SellStartDate < dateSince // This criterion can be passed to the database by LINQ to
Entities
            select ...;

List<Product> products = query.ToList();
```

Considerations

- In some cases, you can improve performance by partitioning data horizontally. If different operations access different attributes of the data, horizontal partitioning may reduce contention. Often, most operations are run against a small subset of the data, so spreading this load may improve performance. See Data partitioning.
- For operations that have to support unbounded queries, implement pagination and only fetch a limited number of entities at a time. For example, if a customer is browsing a product catalog, you can show one page of results at a time.
- When possible, take advantage of features built into the data store. For example, SQL databases typically provide aggregate functions.
- If you're using a data store that doesn't support a particular function, such as aggregation, you could store the calculated result elsewhere, updating the value as records are added or updated, so the application doesn't have to recalculate the value each time it's needed.
- If you see that requests are retrieving a large number of fields, examine the source code to determine whether all of these fields are necessary. Sometimes these requests are the result of poorly designed `SELECT *` query.
- Similarly, requests that retrieve a large number of entities may be sign that the application is not filtering data correctly. Verify that all of these entities are needed. Use database-side filtering if possible, for example, by using `WHERE` clauses in SQL.
- Offloading processing to the database is not always the best option. Only use this strategy when the database is designed or optimized to do so. Most database systems are highly optimized for certain functions, but are not designed to act as general-purpose application engines. For more information, see the Busy Database antipattern.

How to detect extraneous fetching antipattern

Symptoms of extraneous fetching include high latency and low throughput. If the data is retrieved from a data store, increased contention is also probable. End users are likely to report extended response times or failures caused by services timing out. These failures could return HTTP 500 (Internal Server) errors or HTTP 503 (Service Unavailable) errors. Examine the event logs for the web server, which likely contain more detailed information about the causes and circumstances of the errors.

The symptoms of this antipattern and some of the telemetry obtained might be very similar to those of the Monolithic Persistence antipattern.

You can perform the following steps to help identify the cause:

1. Identify slow workloads or transactions by performing load-testing, process monitoring, or other methods of capturing instrumentation data.
2. Observe any behavioral patterns exhibited by the system. Are there particular limits in terms of transactions per second or volume of users?

3. Correlate the instances of slow workloads with behavioral patterns.
4. Identify the data stores being used. For each data source, run lower-level telemetry to observe the behavior of operations.
5. Identify any slow-running queries that reference these data sources.
6. Perform a resource-specific analysis of the slow-running queries and ascertain how the data is used and consumed.

Look for any of these symptoms:

- Frequent, large I/O requests made to the same resource or data store.
- Contention in a shared resource or data store.
- An operation that frequently receives large volumes of data over the network.
- Applications and services spending significant time waiting for I/O to complete.

Example diagnosis

The following sections apply these steps to the previous examples.

Identify slow workloads

This graph shows performance results from a load test that simulated up to 400 concurrent users running the `GetAllFieldsAsync` method shown earlier. Throughput diminishes slowly as the load increases. Average response time goes up as the workload increases.

A load test for the `AggregateOnClientAsync` operation shows a similar pattern. The volume of requests is reasonably stable. The average response time increases with the workload, although more slowly than the previous graph.

Correlate slow workloads with behavioral patterns

Any correlation between regular periods of high usage and slowing performance can indicate areas of concern. Closely examine the performance profile of functionality that is suspected to be slow running, to determine whether it matches the load testing performed earlier.

Load test the same functionality using step-based user loads, to find the point where performance drops significantly or fails completely. If that point falls within the bounds of your expected real-world usage, examine how the functionality is implemented.

A slow operation is not necessarily a problem, if it is not being performed when the system is under stress, is not time critical, and does not negatively affect the performance of other important operations. For example, generating monthly operational statistics might be a long-running operation, but it can probably be performed as a batch process and run as a low-priority job. On the other hand, customers querying the product catalog is a critical business operation. Focus on the telemetry generated by these critical operations to see how the performance varies during periods of high usage.

Identify data sources in slow workloads

If you suspect that a service is performing poorly because of the way it retrieves data, investigate how the application interacts with the repositories it uses. Monitor the live system to see which sources are accessed during periods of poor performance.

For each data source, instrument the system to capture the following:

- The frequency that each data store is accessed.
- The volume of data entering and exiting the data store.
- The timing of these operations, especially the latency of requests.
- The nature and rate of any errors that occur while accessing each data store under typical load.

Compare this information against the volume of data being returned by the application to the client. Track the ratio of the volume of data returned by the data store against the volume of data returned to the client. If there is any large disparity, investigate to determine whether the application is fetching data that it doesn't need.

You may be able to capture this data by observing the live system and tracing the lifecycle of each user request,

or you can model a series of synthetic workloads and run them against a test system.

The following graphs show telemetry captured using New Relic APM during a load test of the `GetAllFieldsAsync` method. Note the difference between the volumes of data received from the database and the corresponding HTTP responses.

For each request, the database returned 80,503 bytes, but the response to the client only contained 19,855 bytes, about 25% of the size of the database response. The size of the data returned to the client can vary depending on the format. For this load test, the client requested JSON data. Separate testing using XML (not shown) had a response size of 35,655 bytes, or 44% of the size of the database response.

The load test for the `AggregateOnClientAsync` method shows more extreme results. In this case, each test performed a query that retrieved over 280 Kb of data from the database, but the JSON response was a mere 14 bytes. The wide disparity is because the method calculates an aggregated result from a large volume of data.

Identify and analyze slow queries

Look for database queries that consume the most resources and take the most time to execute. You can add instrumentation to find the start and completion times for many database operations. Many data stores also provide in-depth information on how queries are performed and optimized. For example, the Query Performance pane in the Azure SQL Database management portal lets you select a query and view detailed runtime performance information. Here is the query generated by the `GetAllFieldsAsync` operation:

```
SELECT
    [Extent1].[ProductID] AS [ProductID],
    [Extent1].[Name] AS [Name],
    [Extent1].[ProductNumber] AS [ProductNumber],
    [Extent1].[MakeFlag] AS [MakeFlag],
    [Extent1].[FinishedGoodsFlag] AS [FinishedGoodsFlag],
    [Extent1].[Color] AS [Color],
    [Extent1].[SafetyStockLevel] AS [SafetyStockLevel],
    [Extent1].[ReorderPoint] AS [ReorderPoint],
    [Extent1].[StandardCost] AS [StandardCost],
    [Extent1].[ListPrice] AS [ListPrice],
    [Extent1].[Size] AS [Size],
    [Extent1].[SizeUnitMeasureCode] AS [SizeUnitMeasureCode],
    [Extent1].[WeightUnitMeasureCode] AS [WeightUnitMeasureCode],
    [Extent1].[Weight] AS [Weight],
```

Query Plan Details Query Plan

Resource Use

Resource	Total / sec	Total	Last Run	Minimum	Maximum
Duration (ms)	117	63450	40	0	3867
CPU (ms)	1	752	0	0	3
Logical Reads	25	13872	16	16	16
Physical Reads	0	0	0	0	0
Logical Writes	0	0	0	0	0

Plan Information

Run Count	867
Last Run Time	03/03/2015 15:32:25
Plan Generation Count	1
Time Plan Cached	03/03/2015 15:26:32

Advanced Information

Plan Handle	0x0600E0004D00CD02A0EDD87D550000000100000000000000000000000
SQL Handle	0x020000004D00CD02B85696A75DA7BEBF79E79259988F30C300000000
Query Hash	0x05ACF4F9056ACADC
Query Plan Hash	0x0DA11AA10A268A7B

Implement the solution and verify the result

After changing the `GetRequiredFieldsAsync` method to use a SELECT statement on the database side, load testing showed the following results.

This load test used the same deployment and the same simulated workload of 400 concurrent users as before. The graph shows much lower latency. Response time rises with load to approximately 1.3 seconds, compared to 4 seconds in the previous case. The throughput is also higher at 350 requests per second compared to 100 earlier. The volume of data retrieved from the database now closely matches the size of the HTTP response messages.

Load testing using the `AggregateOnDatabaseAsync` method generates the following results:

The average response time is now minimal. This is an order of magnitude improvement in performance, caused primarily by the large reduction in I/O from the database.

Here is the corresponding telemetry for the `AggregateOnDatabaseAsync` method. The amount of data retrieved from the database was vastly reduced, from over 280 Kb per transaction to 53 bytes. As a result, the maximum sustained number of requests per minute was raised from around 2,000 to over 25,000.

Related resources

- Busy Database antipattern
- Chatty I/O antipattern
- Data partitioning best practices

Improper Instantiation antipattern

11/19/2021 • 7 minutes to read • Edit Online

Sometimes new instances of a class are continually created, when it is meant to be created once and then shared. This behavior can hurt performance, and is called an *improper instantiation antipattern*. An antipattern is a common response to a recurring problem that is usually ineffective and may even be counter-productive.

Problem description

Many libraries provide abstractions of external resources. Internally, these classes typically manage their own connections to the resource, acting as brokers that clients can use to access the resource. Here are some examples of broker classes that are relevant to Azure applications:

- `System.Net.Http.HttpClient`. Communicates with a web service using HTTP.
- `Microsoft.ServiceBus.Messaging.QueueClient`. Posts and receives messages to a Service Bus queue.
- `Microsoft.Azure.Documents.Client.DocumentClient`. Connects to a Cosmos DB instance.
- `StackExchange.Redis.ConnectionMultiplexer`. Connects to Redis, including Azure Cache for Redis.

These classes are intended to be instantiated once and reused throughout the lifetime of an application. However, it's a common misunderstanding that these classes should be acquired only as necessary and released quickly. (The ones listed here happen to be .NET libraries, but the pattern is not unique to .NET.) The following ASP.NET example creates an instance of `HttpClient` to communicate with a remote service. You can find the complete sample here.

```
public class NewHttpClientInstancePerRequestController : ApiController
{
    // This method creates a new instance of HttpClient and disposes it for every call to GetProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        using (var httpClient = new HttpClient())
        {
            var hostName = HttpContext.Current.Request.Url.Host;
            var result = await httpClient.GetStringAsync(string.Format("http://{0}:8080/api/...",
hostName));
            return new Product { Name = result };
        }
    }
}
```

In a web application, this technique is not scalable. A new `HttpClient` object is created for each user request. Under heavy load, the web server may exhaust the number of available sockets, resulting in `SocketException` errors.

This problem is not restricted to the `HttpClient` class. Other classes that wrap resources or are expensive to create might cause similar issues. The following example creates an instance of the `ExpensiveToCreateService` class. Here the issue is not necessarily socket exhaustion, but simply how long it takes to create each instance. Continually creating and destroying instances of this class might adversely affect the scalability of the system.

```
public class NewServiceInstancePerRequestController : ApiController
{
    public async Task<Product> GetProductAsync(string id)
    {
        var expensiveToCreateService = new ExpensiveToCreateService();
        return await expensiveToCreateService.GetProductByIdAsync(id);
    }
}

public class ExpensiveToCreateService
{
    public ExpensiveToCreateService()
    {
        // Simulate delay due to setup and configuration of ExpensiveToCreateService
        Thread.SpinWait(Int32.MaxValue / 100);
    }
    ...
}
```

How to fix improper instantiation antipattern

If the class that wraps the external resource is shareable and thread-safe, create a shared singleton instance or a pool of reusable instances of the class.

The following example uses a static `HttpClient` instance, thus sharing the connection across all requests.

```
public class SingleHttpClientInstanceController : ApiController
{
    private static readonly HttpClient httpClient;

    static SingleHttpClientInstanceController()
    {
        httpClient = new HttpClient();
    }

    // This method uses the shared instance of HttpClient for every call to GetProductAsync.
    public async Task<Product> GetProductAsync(string id)
    {
        var hostName = HttpContext.Current.Request.Url.Host;
        var result = await httpClient.GetStringAsync(string.Format("http://{0}:8080/api/...", hostName));
        return new Product { Name = result };
    }
}
```

Considerations

- The key element of this antipattern is repeatedly creating and destroying instances of a *shareable* object. If a class is not shareable (not thread-safe), then this antipattern does not apply.
- The type of shared resource might dictate whether you should use a singleton or create a pool. The `HttpClient` class is designed to be shared rather than pooled. Other objects might support pooling, enabling the system to spread the workload across multiple instances.
- Objects that you share across multiple requests *must* be thread-safe. The `HttpClient` class is designed to be used in this manner, but other classes might not support concurrent requests, so check the available documentation.
- Be careful about setting properties on shared objects, as this can lead to race conditions. For example, setting `DefaultRequestHeaders` on the `HttpClient` class before each request can create a race condition. Set such properties once (for example, during startup), and create separate instances if you need to

configure different settings.

- Some resource types are scarce and should not be held onto. Database connections are an example. Holding an open database connection that is not required may prevent other concurrent users from gaining access to the database.
- In the .NET Framework, many objects that establish connections to external resources are created by using static factory methods of other classes that manage these connections. These objects are intended to be saved and reused, rather than disposed and re-created. For example, in Azure Service Bus, the `QueueClient` object is created through a `MessagingFactory` object. Internally, the `MessagingFactory` manages connections. For more information, see Best Practices for performance improvements using Service Bus Messaging.

How to detect improper instantiation antipattern

Symptoms of this problem include a drop in throughput or an increased error rate, along with one or more of the following:

- An increase in exceptions that indicate exhaustion of resources such as sockets, database connections, file handles, and so on.
- Increased memory use and garbage collection.
- An increase in network, disk, or database activity.

You can perform the following steps to help identify this problem:

1. Performing process monitoring of the production system, to identify points when response times slow down or the system fails due to lack of resources.
2. Examine the telemetry data captured at these points to determine which operations might be creating and destroying resource-consuming objects.
3. Load test each suspected operation, in a controlled test environment rather than the production system.
4. Review the source code and examine the how broker objects are managed.

Look at stack traces for operations that are slow-running or that generate exceptions when the system is under load. This information can help to identify how these operations are using resources. Exceptions can help to determine whether errors are caused by shared resources being exhausted.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Identify points of slowdown or failure

The following image shows results generated using New Relic APM, showing operations that have a poor response time. In this case, the `GetProductAsync` method in the `NewHttpClientInstancePerRequest` controller is worth investigating further. Notice that the error rate also increases when these operations are running.

Examine telemetry data and find correlations

The next image shows data captured using thread profiling, over the same period corresponding as the previous image. The system spends a significant time opening socket connections, and even more time closing them and handling socket exceptions.

Performing load testing

Use load testing to simulate the typical operations that users might perform. This can help to identify which parts of a system suffer from resource exhaustion under varying loads. Perform these tests in a controlled environment rather than the production system. The following graph shows the throughput of requests handled by the `NewHttpClientInstancePerRequest` controller as the user load increases to 100 concurrent users.

At first, the volume of requests handled per second increases as the workload increases. At about 30 users, however, the volume of successful requests reaches a limit, and the system starts to generate exceptions. From then on, the volume of exceptions gradually increases with the user load.

The load test reported these failures as HTTP 500 (Internal Server) errors. Reviewing the telemetry showed that these errors were caused by the system running out of socket resources, as more and more `HttpClient` objects were created.

The next graph shows a similar test for a controller that creates the custom `ExpensiveToCreateService` object.

This time, the controller does not generate any exceptions, but throughput still reaches a plateau, while the average response time increases by a factor of 20. (The graph uses a logarithmic scale for response time and throughput.) Telemetry showed that creating new instances of the `ExpensiveToCreateService` was the main cause of the problem.

Implement the solution and verify the result

After switching the `GetProductAsync` method to share a single `HttpClient` instance, a second load test showed improved performance. No errors were reported, and the system was able to handle an increasing load of up to 500 requests per second. The average response time was cut in half, compared with the previous test.

For comparison, the following image shows the stack trace telemetry. This time, the system spends most of its time performing real work, rather than opening and closing sockets.

The next graph shows a similar load test using a shared instance of the `ExpensiveToCreateService` object. Again, the volume of handled requests increases in line with the user load, while the average response time remains

low.

Creating a Single Service Instance
User Load
Throughput/Sec and Response Time (s)
100
90
80
70
60
50
40
30
20
10
0
1000
100
10
1
0.1
0.01
00:00
00:15
00:30
00:45
01:00
01:15
01:30
01:45
02:00
02:15
02:30
02:45
03:00
03:15
03:30
03:45
04:00
04:15
04:30
04:45
05:00
05:15
05:30
05:45
06:00
06:15
06:30
06:45
07:00
07:15
07:30
07:45
08:00
08:15
08:30
08:45
09:00
09:15
09:30
09:45
10:00
User Load
Requests /Sec
Errors/Sec
Avg. Response Time

Monolithic Persistence antipattern

11/19/2021 • 6 minutes to read • Edit Online

Putting all of an application's data into a single data store can hurt performance, either because it leads to resource contention, or because the data store is not a good fit for some of the data.

Problem description

Historically, applications have often used a single data store, regardless of the different types of data that the application might need to store. Usually this was done to simplify the application design, or else to match the existing skill set of the development team.

Modern cloud-based systems often have additional functional and nonfunctional requirements, and need to store many heterogeneous types of data, such as documents, images, cached data, queued messages, application logs, and telemetry. Following the traditional approach and putting all of this information into the same data store can hurt performance, for two main reasons:

- Storing and retrieving large amounts of unrelated data in the same data store can cause contention, which in turn leads to slow response times and connection failures.
- Whichever data store is chosen, it might not be the best fit for all of the different types of data, or it might not be optimized for the operations that the application performs.

The following example shows an ASP.NET Web API controller that adds a new record to a database and also records the result to a log. The log is held in the same database as the business data. You can find the complete sample here.

```
public class MonoController : ApiController
{
    private static readonly string ProductionDb = ...;

    public async Task<IHttpActionResult> PostAsync([FromBody]string value)
    {
        await DataAccess.InsertPurchaseOrderHeaderAsync(ProductionDb);
        await DataAccess.LogAsync(ProductionDb, LogTableName);
        return Ok();
    }
}
```

The rate at which log records are generated will probably affect the performance of the business operations. And if another component, such as an application process monitor, regularly reads and processes the log data, that can also affect the business operations.

How to fix the problem

Separate data according to its use. For each data set, select a data store that best matches how that data set will be used. In the previous example, the application should be logging to a separate store from the database that holds business data:

```
public class PolyController : ApiController
{
    private static readonly string ProductionDb = ...;
    private static readonly string LogDb = ...;

    public async Task<IHttpActionResult> PostAsync([FromBody]string value)
    {
        await DataAccess.InsertPurchaseOrderHeaderAsync(ProductionDb);
        // Log to a different data store.
        await DataAccess.LogAsync(LogDb, LogTableName);
        return Ok();
    }
}
```

Considerations

- Separate data by the way it is used and how it is accessed. For example, don't store log information and business data in the same data store. These types of data have significantly different requirements and patterns of access. Log records are inherently sequential, while business data is more likely to require random access, and is often relational.
- Consider the data access pattern for each type of data. For example, store formatted reports and documents in a document database such as Cosmos DB, but use Azure Cache for Redis to cache temporary data.
- If you follow this guidance but still reach the limits of the database, you may need to scale up the database. Also consider scaling horizontally and partitioning the load across database servers. However, partitioning may require redesigning the application. For more information, see Data partitioning.

How to detect the problem

The system will likely slow down dramatically and eventually fail, as the system runs out of resources such as database connections.

You can perform the following steps to help identify the cause.

1. Instrument the system to record the key performance statistics. Capture timing information for each operation, as well as the points where the application reads and writes data.
2. If possible, monitor the system running for a few days in a production environment to get a real-world view of how the system is used. If this is not possible, run scripted load tests with a realistic volume of virtual users performing a typical series of operations.
3. Use the telemetry data to identify periods of poor performance.
4. Identify which data stores were accessed during those periods.
5. Identify data storage resources that might be experiencing contention.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Instrument and monitor the system

The following graph shows the results of load testing the sample application described earlier. The test used a step load of up to 1000 concurrent users.

As the load increases to 700 users, so does the throughput. But at that point, throughput levels off, and the system appears to be running at its maximum capacity. The average response gradually increases with user load, showing that the system can't keep up with demand.

Identify periods of poor performance

If you are monitoring the production system, you might notice patterns. For example, response times might drop off significantly at the same time each day. This could be caused by a regular workload or scheduled batch job, or just because the system has more users at certain times. You should focus on the telemetry data for these events.

Look for correlations between increased response times and increased database activity or I/O to shared resources. If there are correlations, it means the database might be a bottleneck.

Identify which data stores are accessed during those periods

The next graph shows the utilization of database throughput units (DTU) during the load test. (A DTU is a measure of available capacity, and is a combination of CPU utilization, memory allocation, I/O rate.) Utilization of DTUs quickly reached 100%. This is roughly the point where throughput peaked in the previous graph. Database utilization remained very high until the test finished. There is a slight drop toward the end, which could be caused by throttling, competition for database connections, or other factors.

Examine the telemetry for the data stores

Instrument the data stores to capture the low-level details of the activity. In the sample application, the data access statistics showed a high volume of insert operations performed against both the `PurchaseOrderHeader` table and the `MonoLog` table.

Query	Run Count	CPU ms/sec	Duration ms/sec
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 +1 , ((CASE eq1.statement_end	2	23	25
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 +1 , ((CASE eq1.statement_end	1	14	16
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 +1 , ((CASE eq1.statement_end	2	13	14
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 +1 , ((CASE eq1.statement_end	2	12	12
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 +1 , ((CASE eq1.statement_end	1	11	12
INSERT INTO Purchasing.PurchaseOrderHeader (RevisionNumber, Status, EmployeeID, VendorID, !	21662	7	2155
select @UsedPageCount = ISNULL(SUM(reserved_page_count), 0) FROM sys.dm_db_partition_stat	2	6	14
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	13789	2	1094
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	10579	1	832
SELECT database_id FROM sys.databases WHERE name=N'AdventureWorks2012' and db_name()=	18	1	1
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	896	0	72
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	97	0	11
INSERT INTO dbo.MonoLog(LogId, Message, LogTime) VALUES(@LogId, @Message, @LogTime)	13	0	1
SELECT dtb.collation_name AS [Database_Collation], dtb.create_date AS [Database_CreationDate],	2	0	0
SELECT st.name AS [FederationSystemDataType_Name], st.user_type_id AS [FederationSystemData	2	0	0
select is_federation_member from sys.databases where name = N'AdventureWorks2012'	4	0	0
SELECT CAST(st.system_type_id AS int) AS [SystemDataType_BaseDataType], st.is_nullable AS [Syst	2	0	0
select is_federation_member from sys.databases where name = N'AdventureWorks2012'	1	0	0
SELECT CAST(CASE WHEN btst.name IN ('varchar', 'nvarchar', 'varbinary') and st.max_length = -1 T	2	0	0
SELECT fed.name AS [Federation_Name], fed.federation_id AS [Federation_ID], DB_NAME() AS [Fec	2	0	0
SELECT CAST((select SUBSTRING(text,eq1.statement_start_offset/2 +1 , ((CASE eq1.statement_end	1	0	0
INSERT INTO Purchasing.PurchaseOrderHeader (RevisionNumber, Status, EmployeeID, VendorID, !	1	0	0

Identify resource contention

At this point, you can review the source code, focusing on the points where contended resources are accessed by the application. Look for situations such as:

- Data that is logically separate being written to the same store. Data such as logs, reports, and queued messages should not be held in the same database as business information.
- A mismatch between the choice of data store and the type of data, such as large blobs or XML documents in a relational database.
- Data with significantly different usage patterns that share the same store, such as high-write/low-read data being stored with low-write/high-read data.

Implement the solution and verify the result

The application was changed to write logs to a separate data store. Here are the load test results:

The pattern of throughput is similar to the earlier graph, but the point at which performance peaks is approximately 500 requests per second higher. The average response time is marginally lower. However, these statistics don't tell the full story. Telemetry for the business database shows that DTU utilization peaks at around 75%, rather than 100%.

Similarly, the maximum DTU utilization of the log database only reaches about 70%. The databases are no longer the limiting factor in the performance of the system.

Related resources

- Choose the right data store
- Criteria for choosing a data store
- Data Access for Highly Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence
- Data partitioning

No Caching antipattern

11/19/2021 • 8 minutes to read • Edit Online

Anti-patterns are common design flaws that can break your software or applications under stress situations and should not be overlooked. A *no caching antipattern* occurs when a cloud application that handles many concurrent requests, repeatedly fetches the same data. This can reduce performance and scalability.

When data is not cached, it can cause a number of undesirable behaviors, including:

- Repeatedly fetching the same information from a resource that is expensive to access, in terms of I/O overhead or latency.
- Repeatedly constructing the same objects or data structures for multiple requests.
- Making excessive calls to a remote service that has a service quota and throttles clients past a certain limit.

In turn, these problems can lead to poor response times, increased contention in the data store, and poor scalability.

Examples of no caching antipattern

The following example uses Entity Framework to connect to a database. Every client request results in a call to the database, even if multiple requests are fetching exactly the same data. The cost of repeated requests, in terms of I/O overhead and data access charges, can accumulate quickly.

```
public class PersonRepository : IPersonRepository
{
    public async Task<Person> GetAsync(int id)
    {
        using (var context = new AdventureWorksContext())
        {
            return await context.People
                .Where(p => p.Id == id)
                .FirstOrDefaultAsync()
                .ConfigureAwait(false);
        }
    }
}
```

You can find the complete sample here.

This antipattern typically occurs because:

- Not using a cache is simpler to implement, and it works fine under low loads. Caching makes the code more complicated.
- The benefits and drawbacks of using a cache are not clearly understood.
- There is concern about the overhead of maintaining the accuracy and freshness of cached data.
- An application was migrated from an on-premises system, where network latency was not an issue, and the system ran on expensive high-performance hardware, so caching wasn't considered in the original design.
- Developers aren't aware that caching is a possibility in a given scenario. For example, developers may not think of using ETags when implementing a web API.

How to fix the no caching antipattern

The most popular caching strategy is the *on-demand* or *cache-aside* strategy.

- On read, the application tries to read the data from the cache. If the data isn't in the cache, the application retrieves it from the data source and adds it to the cache.
- On write, the application writes the change directly to the data source and removes the old value from the cache. It will be retrieved and added to the cache the next time it is required.

This approach is suitable for data that changes frequently. Here is the previous example updated to use the Cache-Aside pattern.

```
public class CachedPersonRepository : IPersonRepository
{
    private readonly PersonRepository _innerRepository;

    public CachedPersonRepository(PersonRepository innerRepository)
    {
        _innerRepository = innerRepository;
    }

    public async Task<Person> GetAsync(int id)
    {
        return await CacheService.GetAsync<Person>("p:" + id, () =>
_innerRepository.GetAsync(id)).ConfigureAwait(false);
    }
}

public class CacheService
{
    private static ConnectionMultiplexer _connection;

    public static async Task<T> GetAsync<T>(string key, Func<Task<T>> loadCache, double
expirationTimeInMinutes)
    {
        IDatabase cache = Connection.GetDatabase();
        T value = await GetAsync<T>(cache, key).ConfigureAwait(false);
        if (value == null)
        {
            // Value was not found in the cache. Call the lambda to get the value from the database.
            value = await loadCache().ConfigureAwait(false);
            if (value != null)
            {
                // Add the value to the cache.
                await SetAsync(cache, key, value, expirationTimeInMinutes).ConfigureAwait(false);
            }
        }
        return value;
    }
}
```

Notice that the `GetAsync` method now calls the `CacheService` class, rather than calling the database directly. The `CacheService` class first tries to get the item from Azure Cache for Redis. If the value isn't found in the cache, the `CacheService` invokes a lambda function that was passed to it by the caller. The lambda function is responsible for fetching the data from the database. This implementation decouples the repository from the particular caching solution, and decouples the `CacheService` from the database.

Considerations for caching strategy

- If the cache is unavailable, perhaps because of a transient failure, don't return an error to the client. Instead, fetch the data from the original data source. However, be aware that while the cache is being recovered, the original data store could be swamped with requests, resulting in timeouts and failed connections. (After all, this is one of the motivations for using a cache in the first place.) Use a technique such as the Circuit Breaker pattern to avoid overwhelming the data source.

- Applications that cache dynamic data should be designed to support eventual consistency.
- For web APIs, you can support client-side caching by including a Cache-Control header in request and response messages, and using ETags to identify versions of objects. For more information, see API implementation.
- You don't have to cache entire entities. If most of an entity is static but only a small piece changes frequently, cache the static elements and retrieve the dynamic elements from the data source. This approach can help to reduce the volume of I/O being performed against the data source.
- In some cases, if volatile data is short-lived, it can be useful to cache it. For example, consider a device that continually sends status updates. It might make sense to cache this information as it arrives, and not write it to a persistent store at all.
- To prevent data from becoming stale, many caching solutions support configurable expiration periods, so that data is automatically removed from the cache after a specified interval. You may need to tune the expiration time for your scenario. Data that is highly static can stay in the cache for longer periods than volatile data that may become stale quickly.
- If the caching solution doesn't provide built-in expiration, you may need to implement a background process that occasionally sweeps the cache, to prevent it from growing without limits.
- Besides caching data from an external data source, you can use caching to save the results of complex computations. Before you do that, however, instrument the application to determine whether the application is really CPU bound.
- It might be useful to prime the cache when the application starts. Populate the cache with the data that is most likely to be used.
- Always include instrumentation that detects cache hits and cache misses. Use this information to tune caching policies, such what data to cache, and how long to hold data in the cache before it expires.
- If the lack of caching is a bottleneck, then adding caching may increase the volume of requests so much that the web front end becomes overloaded. Clients may start to receive HTTP 503 (Service Unavailable) errors. These are an indication that you should scale out the front end.

How to detect a no caching antipattern

You can perform the following steps to help identify whether lack of caching is causing performance problems:

1. Review the application design. Take an inventory of all the data stores that the application uses. For each, determine whether the application is using a cache. If possible, determine how frequently the data changes. Good initial candidates for caching include data that changes slowly, and static reference data that is read frequently.
2. Instrument the application and monitor the live system to find out how frequently the application retrieves data or calculates information.
3. Profile the application in a test environment to capture low-level metrics about the overhead associated with data access operations or other frequently performed calculations.
4. Perform load testing in a test environment to identify how the system responds under a normal workload and under heavy load. Load testing should simulate the pattern of data access observed in the production environment using realistic workloads.
5. Examine the data access statistics for the underlying data stores and review how often the same data requests are repeated.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Instrument the application and monitor the live system

Instrument the application and monitor it to get information about the specific requests that users make while the application is in production.

The following image shows monitoring data captured by New Relic during a load test. In this case, the only HTTP GET operation performed is `Person/GetAsync`. But in a live production environment, knowing the relative frequency that each request is performed can give you insight into which resources should be cached.

If you need a deeper analysis, you can use a profiler to capture low-level performance data in a test environment (not the production system). Look at metrics such as I/O request rates, memory usage, and CPU utilization. These metrics may show a large number of requests to a data store or service, or repeated processing that performs the same calculation.

Load test the application

The following graph shows the results of load testing the sample application. The load test simulates a step load of up to 800 users performing a typical series of operations.

The number of successful tests performed each second reaches a plateau, and additional requests are slowed as a result. The average test time steadily increases with the workload. The response time levels off once the user load peaks.

Examine data access statistics

Data access statistics and other information provided by a data store can give useful information, such as which queries are repeated most frequently. For example, in Microsoft SQL Server, the `sys.dm_exec_query_stats` management view has statistical information for recently executed queries. The text for each query is available in the `sys.dm_exec-query_plan` view. You can use a tool such as SQL Server Management Studio to run the following SQL query and determine how frequently queries are performed.

```
SELECT UseCounts, Text, Query_Plan
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)
```

The `UseCount` column in the results indicates how frequently each query is run. The following image shows that the third query was run more than 250,000 times, significantly more than any other query.

```
SELECT UseCounts, Text, Query_Plan
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
CROSS APPLY sys.dm_exec_query_plan(plan_handle)
```

100 %

Results | Messages

	UseCounts	Text	Query_Plan
1	1	SELECT UseCounts, Text, Query_Plan FROM sys.dm_...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
2	1	select is_federation_member from sys.databases where ...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
3	256049	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
4	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
5	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
6	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
7	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
8	3	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
9	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
10	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
11	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
12	3	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
13	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
14	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
15	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
16	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
17	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
18	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
19	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
20	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
21	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
22	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
23	2	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...
24	1	(@p__linq__0 int)SELECT TOP (2) [Extent1].[Busine...	<ShowPlanXML xmlns="http://schemas.microsoft.com...

Here is the SQL query that is causing so many database requests:

```
(@p__linq__0 int)SELECT TOP (2)
[Extent1].[BusinessEntityId] AS [BusinessEntityId],
[Extent1].[FirstName] AS [FirstName],
[Extent1].[LastName] AS [LastName]
FROM [Person].[Person] AS [Extent1]
WHERE [Extent1].[BusinessEntityId] = @p__linq__0
```

This is the query that Entity Framework generates in `GetByIdAsync` method shown earlier.

Implement the cache strategy solution and verify the result

After you incorporate a cache, repeat the load tests and compare the results to the earlier load tests without a cache. Here are the load test results after adding a cache to the sample application.

The volume of successful tests still reaches a plateau, but at a higher user load. The request rate at this load is significantly higher than earlier. Average test time still increases with load, but the maximum response time is 0.05 ms, compared with 1 ms earlier—a 20× improvement.

Related resources

- API implementation best practices
- Cache-Aside pattern
- Caching best practices
- Circuit Breaker pattern

Noisy Neighbor antipattern

11/19/2021 • 5 minutes to read • Edit Online

Multitenant systems share resources between tenants. This means that the activity of one tenant can have a negative impact on another tenant's use of the system.

Problem description

When you build a service to be shared by multiple customers or tenants, you can build it to be *multitenanted*. A benefit of multitenant systems is that resources can be pooled and shared among tenants. This often results in lower costs and improved efficiency. However, if a single tenant uses a disproportionate amount of the resources available in the system, the overall performance of the system can suffer. The *noisy neighbor* problem occurs when one tenant's performance is degraded because of the activities of another tenant.

Consider an example multitenant system with two tenants. Tenant A's usage patterns and tenant B's usage patterns coincide, which means that at peak times, the total resource usage is higher than the capacity of the system:

It's likely that whichever tenant's request arrived first will take precedence, and the other tenant will experience a noisy neighbor problem. Alternatively, both tenants may find their performance suffers.

The noisy neighbor problem also occurs even when each individual tenant is consuming relatively small amounts of the system's capacity, but the collective resource usage of many tenants results in a peak in overall usage:

This can happen when you have multiple tenants that all have similar usage patterns, or where you haven't provisioned sufficient capacity for the collective load on the system.

How to fix the problem

Noisy neighbor problems are an inherent risk in multitenant systems, and it's not possible to completely

eliminate the possibility of being affected by a noisy neighbor. However, there are some steps that both clients and service providers can take to reduce the likelihood of noisy neighbor problems, or to mitigate their effects when they are observed.

Actions that clients can take

- Purchase reserved capacity, if available. For example, when using Cosmos DB, purchase reserved throughput, and when using ExpressRoute, provision separate circuits for environments that are sensitive to performance.
- Migrate to a single-tenant instance of the service, or to a service tier with stronger isolation guarantees. For example, when using Service Bus, migrate to the premium tier, and when using Azure Cache for Redis, provision a standard or premium tier cache.
- Ensure your application handles service throttling, to reduce making unnecessary requests to the service.

Actions that service providers can take

- Monitor the resource usage for your system, both overall and for each tenant. Configure alerts to detect spikes in resource usage, and if possible, configure automation to automatically mitigate known issues by scaling up or out.
- Apply resource governance, to avoid a single tenant that's overwhelming the system and reducing the capacity available to others. This might take the form of quota enforcement, through the Throttling pattern or the Rate Limiting pattern.
- Consider provisioning more infrastructure. This might involve scaling up by upgrading some of your solution components, or it might involve scaling out by provisioning additional shards, if you follow the Sharding pattern, or stamps, if you follow the Deployment Stamps pattern.
- Consider allowing tenants to purchase pre-provisioned or reserved capacity. This provides tenants with more certainty that your solution adequately handles their workload.
- Consider approaches to smooth out the resource usage:
  - If you host multiple instances of your solution, consider re-balancing tenants across the instances or stamps. For example, consider placing tenants with predictable and similar usage patterns, across multiple stamps, to flatten the peaks in their usage.
  - Consider whether you have background processes or resource-intensive workloads that aren't time-sensitive. Run these asynchronously at off-peak times, to preserve your peak resource capacity for time-sensitive workloads.
- Consider whether your services provide controls to mitigate noisy neighbor problems. For example, when using Kubernetes, consider using pod limits, and when using Service Fabric, consider using the built-in governance capabilities.
- If applicable, consider restricting the operations that tenants can perform. For example, prevent tenants from executing operations that will run very large database queries. This mitigates the risk of tenants taking actions that might negatively impact other tenants.
- If applicable, consider providing a Quality of Service (QoS) system. When you apply QoS, you prioritize some processes or workloads ahead of others. By factoring QoS into your design and architecture, you can ensure that high-priority operations take precedence when there's pressure on your resources.

Considerations

- In most cases, individual tenants don't mean to cause noisy neighbor issues. Individual tenants may not even be aware that their workloads cause noisy neighbor issues for others.
- However, it's also possible that tenants use vulnerabilities in shared components to attack a service, either individually or by executing a distributed denial of service (DDoS) attack.
- Regardless of the cause, it's important to treat these problems as resource governance issues, and to apply usage quotas, throttling, and governance controls to mitigate the problem.

> **NOTE**
> Make sure that you tell your clients about any throttling that you apply, or any usage quotas on your service. It's important that they reliably handle failed requests, and that they aren't surprised by any limitations or quotas you apply.

How to detect the problem

From a client's perspective, the noisy neighbor problem typically manifests as failed server requests, or requests that take a long time to complete. In particular, if the same request succeeds at other times and appears to fail randomly, there may be a noisy neighbor issue. Client applications should record telemetry to track the success rate and performance of the requests to services, and the applications should also record baseline performance metrics for comparison purposes.

From a service's perspective, the noisy neighbor issue may appear in several ways:

- Spikes in resource usage. It's important to have a clear understanding of your normal baseline resource usage, and to configure monitoring and alerts to detect spikes in resource usage. Ensure you consider all of the resources that could affect your service's performance or availability. These include metrics like server CPU and memory usage, disk IO, database usage, network traffic, and metrics that are exposed by managed services, such as the number of requests and the synthetic and abstract performance metrics, such as the Azure Cosmos DB request units.
- Failures when performing an operation for a tenant, even when that tenant isn't using a large portion of the system's resources. Such a pattern may indicate that the tenant is a victim of the noisy neighbor problem. Consider tracking the resource consumption by tenant. For example, when using Azure Cosmos DB, consider logging the request units used for each request, and add the tenant's identifier as a dimension to the telemetry, so that you can aggregate the request unit consumption for each tenant.

Related resources

- Architectural considerations for a multitenant solution
- Transient fault handling best practices

Retry Storm antipattern

11/19/2021 • 5 minutes to read • Edit Online

When a service is unavailable or busy, having clients retry their connections too frequently can cause the service to struggle to recover, and can make the problem worse. It also doesn't make sense to retry forever, since requests are typically only valid for a defined period of time.

Problem description

In the cloud, services sometimes experience problems and become unavailable to clients, or have to throttle or rate limit their clients. While it's a good practice for clients to retry failed connections to services, it's important they do not retry too frequently or for too long. Retries within a short period of time are unlikely to succeed since the services likely will not have recovered. Also, services can be put under even more stress when lots of connection attempts are made while they're trying to recover, and repeated connection attempts may even overwhelm the service and make the underlying problem worse.

The following example illustrates a scenario where a client connects to a server-based API. If the request doesn't succeed, then the client retries immediately, and keeps retrying forever. Often this sort of behavior is more subtle than in this example, but the same principle applies.

```
public async Task<string> GetDataFromServer()
{
    while(true)
    {
        var result = await httpClient.GetAsync(string.Format("http://{0}:8080/api/...", hostName));
        if (result.IsSuccessStatusCode) break;
    }

    // ... Process result.
}
```

How to fix the problem

Client applications should follow some best practices to avoid causing a retry storm.

- Cap the number of retry attempts, and don't keep retrying for a long period of time. While it might seem easy to write a `while(true)` loop, you almost certainly don't want to actually retry for a long period of time, since the situation that led to the request being initiated has probably changed. In most applications, retrying for a few seconds or minutes is sufficient.
- Pause between retry attempts. If a service is unavailable, retrying immediately is unlikely to succeed. Gradually increase the amount of time you wait between attempts, for example by using an exponential backoff strategy.
- Gracefully handle errors. If the service isn't responding, consider whether it makes sense to abort the attempt and return an error back to the user or caller of your component. Consider these failure scenarios when designing your application.
- Consider using the Circuit Breaker pattern, which is designed specifically to help avoid retry storms.
- If the server provides a `retry-after` response header, make sure you don't attempt to retry until the specified time period has elapsed.
- Use official SDKs when communicating to Azure services. These SDKs generally have built-in retry policies and protections against causing or contributing to retry storms. If you're communicating with a service that

doesn't have an SDK, or where the SDK doesn't handle retry logic correctly, consider using a library like Polly (for .NET) or retry (for JavaScript) to handle your retry logic correctly and avoid writing the code yourself.

- If you're running in an environment that supports it, use a service mesh (or another abstraction layer) to send outbound calls. Typically these tools, such as Dapr, support retry policies and automatically follow best practices, like backing off after repeated attempts. This approach means you don't have to write retry code yourself.
- Consider batching requests and using request pooling where available. Many SDKs handle request batching and connection pooling on your behalf, which will reduce the total number of outbound connection attempts your application makes, although you still need to be careful not to retry these connections too frequently.

Services should also protect themselves against retry storms.

- Add a gateway layer so you can shut off connections during an incident. This is an example of the Bulkhead pattern. Azure provides many different gateway services for different types of solutions including Front Door, Application Gateway, and API Management.
- Throttle requests at your gateway, which ensures you won't accept so many requests that your back-end components can't continue to operate.
- If you're throttling, send back a `retry-after` header to help clients understand when to reattempt their connections.

Considerations

- Clients should consider the type of error returned. Some error types don't indicate a failure of the service, but instead indicate that the client sent an invalid request. For example, if a client application receives a `400 Bad Request` error response, retrying the same request probably is not going to help since the server is telling you that your request is not valid.
- Clients should consider the length of time that makes sense to reattempt connections. The length of time you should retry for will be driven by your business requirements and whether you can reasonably propagate an error back to a user or caller. In most applications, retrying for a few seconds or minutes is sufficient.

How to detect the problem

From a client's perspective, symptoms of this problem could include very long response or processing times, along with telemetry that indicates repeated attempts to retry the connection.

From a service's perspective, symptoms of this problem could include a large number of requests from one client within a short period of time, or a large number of requests from a single client while recovering from outages. Symptoms could also include difficulty when recovering the service, or ongoing cascading failures of the service right after a fault has been repaired.

Example diagnosis

The following sections illustrate one approach to detecting a potential retry storm, both on the client side and the service side.

Identifying from client telemetry

Azure Application Insights records telemetry from applications and makes the data available for querying and visualization. Outbound connections are tracked as dependencies, and information about them can be accessed and charted to identify when a client is making a large number of outbound requests to the same service.

The following graph was taken from the Metrics tab within the Application Insights portal, and displaying the *Dependency failures* metric split by *Remote dependency name*. This illustrates a scenario where there were a large number (over 21,000) of failed connection attempts to a dependency within a short time.

Identifying from server telemetry

Server applications may be able to detect large numbers of connections from a single client. In the following example, Azure Front Door acts as a gateway for an application, and has been configured to log all requests to a Log Analytics workspace.

The following Kusto query can be executed against Log Analytics. It will identify client IP addresses that have sent large numbers of requests to the application within the last day.

```
AzureDiagnostics
| where ResourceType == "FRONTDOORS" and Category == "FrontdoorAccessLog"
| where TimeGenerated > ago(1d)
| summarize count() by bin(TimeGenerated, 1h), clientIp_s
| order by count_ desc
```

Executing this query during a retry storm shows a large number of connection attempts from a single IP address.

Related resources

- Retry pattern
- Circuit Breaker pattern
- Transient fault handling best practices
- Service-specific retry guidance

Synchronous I/O antipattern

11/19/2021 • 6 minutes to read • Edit Online

Blocking the calling thread while I/O completes can reduce performance and affect vertical scalability.

Problem description

A synchronous I/O operation blocks the calling thread while the I/O completes. The calling thread enters a wait state and is unable to perform useful work during this interval, wasting processing resources.

Common examples of I/O include:

- Retrieving or persisting data to a database or any type of persistent storage.
- Sending a request to a web service.
- Posting a message or retrieving a message from a queue.
- Writing to or reading from a local file.

This antipattern typically occurs because:

- It appears to be the most intuitive way to perform an operation.
- The application requires a response from a request.
- The application uses a library that only provides synchronous methods for I/O.
- An external library performs synchronous I/O operations internally. A single synchronous I/O call can block an entire call chain.

The following code uploads a file to Azure blob storage. There are two places where the code blocks waiting for synchronous I/O, the `CreateIfNotExists` method and the `UploadFromStream` method.

```
var blobClient = storageAccount.CreateCloudBlobClient();
var container = blobClient.GetContainerReference("uploadedfiles");

container.CreateIfNotExists();
var blockBlob = container.GetBlockBlobReference("myblob");

// Create or overwrite the "myblob" blob with contents from a local file.
using (var fileStream = File.OpenRead(HostingEnvironment.MapPath("~/FileToUpload.txt")))
{
    blockBlob.UploadFromStream(fileStream);
}
```

Here's an example of waiting for a response from an external service. The `GetUserProfile` method calls a remote service that returns a `UserProfile`.

```
public interface IUserProfileService
{
    UserProfile GetUserProfile();
}

public class SyncController : ApiController
{
    private readonly IUserProfileService _userProfileService;

    public SyncController()
    {
        _userProfileService = new FakeUserProfileService();
    }

    // This is a synchronous method that calls the synchronous GetUserProfile method.
    public UserProfile GetUserProfile()
    {
        return _userProfileService.GetUserProfile();
    }
}
```

You can find the complete code for both of these examples here.

How to fix the problem

Replace synchronous I/O operations with asynchronous operations. This frees the current thread to continue performing meaningful work rather than blocking, and helps improve the utilization of compute resources. Performing I/O asynchronously is particularly efficient for handling an unexpected surge in requests from client applications.

Many libraries provide both synchronous and asynchronous versions of methods. Whenever possible, use the asynchronous versions. Here is the asynchronous version of the previous example that uploads a file to Azure blob storage.

```
var blobClient = storageAccount.CreateCloudBlobClient();
var container = blobClient.GetContainerReference("uploadedfiles");

await container.CreateIfNotExistsAsync();

var blockBlob = container.GetBlockBlobReference("myblob");

// Create or overwrite the "myblob" blob with contents from a local file.
using (var fileStream = File.OpenRead(HostingEnvironment.MapPath("~/FileToUpload.txt")))
{
    await blockBlob.UploadFromStreamAsync(fileStream);
}
```

The `await` operator returns control to the calling environment while the asynchronous operation is performed. The code after this statement acts as a continuation that runs when the asynchronous operation has completed.

A well designed service should also provide asynchronous operations. Here is an asynchronous version of the web service that returns user profiles. The `GetUserProfileAsync` method depends on having an asynchronous version of the User Profile service.

```
public interface IUserProfileService
{
    Task<UserProfile> GetUserProfileAsync();
}

public class AsyncController : ApiController
{
    private readonly IUserProfileService _userProfileService;

    public AsyncController()
    {
        _userProfileService = new FakeUserProfileService();
    }

    // This is an synchronous method that calls the Task based GetUserProfileAsync method.
    public Task<UserProfile> GetUserProfileAsync()
    {
        return _userProfileService.GetUserProfileAsync();
    }
}
```

For libraries that don't provide asynchronous versions of operations, it may be possible to create asynchronous wrappers around selected synchronous methods. Follow this approach with caution. While it may improve responsiveness on the thread that invokes the asynchronous wrapper, it actually consumes more resources. An extra thread may be created, and there is overhead associated with synchronizing the work done by this thread. Some tradeoffs are discussed in this blog post: Should I expose asynchronous wrappers for synchronous methods?

Here is an example of an asynchronous wrapper around a synchronous method.

```
// Asynchronous wrapper around synchronous library method
private async Task<int> LibraryIOOperationAsync()
{
    return await Task.Run(() => LibraryIOOperation());
}
```

Now the calling code can await on the wrapper:

```
// Invoke the asynchronous wrapper using a task
await LibraryIOOperationAsync();
```

Considerations

- I/O operations that are expected to be very short lived and are unlikely to cause contention might be more performant as synchronous operations. An example might be reading small files on an SSD drive. The overhead of dispatching a task to another thread, and synchronizing with that thread when the task completes, might outweigh the benefits of asynchronous I/O. However, these cases are relatively rare, and most I/O operations should be done asynchronously.
- Improving I/O performance may cause other parts of the system to become bottlenecks. For example, unblocking threads might result in a higher volume of concurrent requests to shared resources, leading in turn to resource starvation or throttling. If that becomes a problem, you might need to scale out the number of web servers or partition data stores to reduce contention.

How to detect the problem

For users, the application may seem unresponsive periodically. The application might fail with timeout

exceptions. These failures could also return HTTP 500 (Internal Server) errors. On the server, incoming client requests might be blocked until a thread becomes available, resulting in excessive request queue lengths, manifested as HTTP 503 (Service Unavailable) errors.

You can perform the following steps to help identify the problem:

1. Monitor the production system and determine whether blocked worker threads are constraining throughput.
2. If requests are being blocked due to lack of threads, review the application to determine which operations may be performing I/O synchronously.
3. Perform controlled load testing of each operation that is performing synchronous I/O, to find out whether those operations are affecting system performance.

Example diagnosis

The following sections apply these steps to the sample application described earlier.

Monitor web server performance

For Azure web applications and web roles, it's worth monitoring the performance of the IIS web server. In particular, pay attention to the request queue length to establish whether requests are being blocked waiting for available threads during periods of high activity. You can gather this information by enabling Azure diagnostics. For more information, see:

- Monitor Apps in Azure App Service
- Create and use performance counters in an Azure application

Instrument the application to see how requests are handled once they have been accepted. Tracing the flow of a request can help to identify whether it is performing slow-running calls and blocking the current thread. Thread profiling can also highlight requests that are being blocked.

Load test the application

The following graph shows the performance of the synchronous `GetUserProfile` method shown earlier, under varying loads of up to 4000 concurrent users. The application is an ASP.NET application running in an Azure Cloud Service web role.

The synchronous operation is hard-coded to sleep for 2 seconds, to simulate synchronous I/O, so the minimum

response time is slightly over 2 seconds. When the load reaches approximately 2500 concurrent users, the average response time reaches a plateau, although the volume of requests per second continues to increase. Note that the scale for these two measures is logarithmic. The number of requests per second doubles between this point and the end of the test.

In isolation, it's not necessarily clear from this test whether the synchronous I/O is a problem. Under heavier load, the application may reach a tipping point where the web server can no longer process requests in a timely manner, causing client applications to receive time-out exceptions.

Incoming requests are queued by the IIS web server and handed to a thread running in the ASP.NET thread pool. Because each operation performs I/O synchronously, the thread is blocked until the operation completes. As the workload increases, eventually all of the ASP.NET threads in the thread pool are allocated and blocked. At that point, any further incoming requests must wait in the queue for an available thread. As the queue length grows, requests start to time out.

Implement the solution and verify the result

The next graph shows the results from load testing the asynchronous version of the code.

Throughput is far higher. Over the same duration as the previous test, the system successfully handles a nearly tenfold increase in throughput, as measured in requests per second. Moreover, the average response time is relatively constant and remains approximately 25 times smaller than the previous test.

Responsible Innovation: A Best Practices Toolkit

11/19/2021 • 2 minutes to read • Edit Online

Responsible innovation is a toolkit that helps developers become good stewards for the future of science and its effect on society. This toolkit provides a set of practices in development, for anticipating and addressing the potential negative impacts of technology on people. We are sharing this as an early stage practice for feedback and learning.

Judgment Call

Judgment Call is an award-winning responsible innovation game and team-based activity that puts Microsoft's AI principles of fairness, privacy and security, reliability and safety, transparency, inclusion, and accountability into action. The game provides an easy-to-use method for cultivating stakeholder empathy through *scenario-imagining*. Game participants write product reviews from the perspective of a particular stakeholder, describing what kind of impact and harms the technology could produce from their point of view.

Harms Modeling

Harms Modeling is a framework for product teams, grounded in four core pillars of responsible innovation, that examine how people's lives can be negatively impacted by technology: injuries, denial of consequential services, infringement on human rights, and erosion of democratic & societal structures. Similar to *Security Threat Modeling*, Harms Modeling enables product teams to anticipate potential real-world impacts of technology, which is a cornerstone of responsible development.

Community Jury

Community Jury is a technique that brings together diverse stakeholders impacted by a technology. It is an adaptation of the citizen jury. The stakeholders are provided an opportunity to learn from experts about a project, deliberate together, and give feedback on use cases and product design. This responsible innovation technique allows project teams to collaborate with researchers to identify stakeholder values, and understand the perceptions and concerns of impacted stakeholders.

Judgment Call

11/19/2021 • 5 minutes to read • Edit Online

Judgment Call is an award-winning game and team-based activity that puts Microsoft's AI principles of fairness, privacy and security, reliability and safety, transparency, inclusion, and accountability into action. The game provides an easy-to-use method for cultivating stakeholder empathy by imagining their scenarios. Game participants write product reviews from the perspective of a particular stakeholder, describing what kind of impact and harms the technology could produce from their point of view.

Benefits

Technology builders need practical methods to incorporate ethics in product development, by considering the values of diverse stakeholders and how technology may uphold or not uphold those values. The goal of the game is to imagine potential outcomes of a product or platform by gaining a better understanding of stakeholders, and what they need and expect.

The game helps people discuss challenging topics in a safe space within the context of gameplay, and gives technology creator a vocabulary to facilitate ethics discussions during product development. It gives managers and designers an interactive tool to lead ethical dialogue with their teams to incorporate ethical design in their products.

The theoretical basis and initial outcomes of the Judgment Call game were presented at the 2019 ACM Design Interactive Systems conference and the game was a finalist in the Fast Company 2019 Innovation by Design Awards (Social Goods category). The game has been presented to thousands of people in the US and internationally. While the largest group facilitated in game play has been over 100, each card deck can involve 1-10 players. This game is not intended to be a substitute for performing robust user research. Rather, it's an exercise to help tech builders learn how to exercise their moral imagination.

Preparation

Judgment Call uses role play to surface ethical concerns in product development so players will anticipate societal impact, write product reviews, and explore mitigations. Players think of what kind of harms and impacts the technology might produce by writing product reviews from the point of view of a stakeholder.

To prepare for this game, download the printable Judgment Call game kit.

During the Game

The players are expected to go through the following steps in this game:

1. **Identify the product**

   Start by identifying a product that your team is building, or a scenario you are working on. For example, if your team is developing synthetic voice technology, your scenario could be developing a voice assistant for online learning.

2. **Pass the cards**

   Once you have decided on the scenario, pass out the cards. Each player receives a stakeholder card, an ethical principle card, a rating card, and a review form. The following is a description of these cards:

- **Stakeholder:** This represents the group of people impacted by the technology or scenario you've chosen. Stakeholders can also include the makers (those proposing and working on the technology) as well as the internal project sponsors (managers and executives who are supportive of the project).
- **Principle:** This includes Microsoft's ethical principles of fairness, privacy and security, reliability and safety, transparency, inclusion, and accountability.
- **Rating:** Star cards give a rating: 1-star is poor, 3-star is mediocre, 5-star is excellent.

3. **Brainstorm and identify stakeholders**

   As a group, brainstorm all the stakeholders that could be directly or indirectly affected by the technology. Then assign each player a stakeholder from the list. In some cases, the Judgment Call decks may have pre-populated stakeholder cards, but players can add additional cards relevant to their situation.

   The deck provides following categories of stakeholders:

   - **Direct:** These use the technology and/or the technology is used on them.
   - **Indirect:** These feel the effects of the technology.
   - **Special populations:** A wide range of categories that includes those who cannot use the technology or choose not to use it, and organizations that may be interested in its deployment like advocacy groups or the government.

   There are a range of harms that can be felt or perpetuated onto each group of stakeholder. Types of stakeholders can include end users of the product, administrators, internal teams, advocacy groups, manufacturers, someone who might be excluded from using the product. Stakeholders can also include the makers who propose and work on the technology, as well as the internal project sponsors, that is, the managers and executives who are supportive of the project.

4. **Moderator selection**

   The moderator describes scenarios for gameplay, guides discussion, and collects comment cards at the end. They will also give an overview of the ethical principles considered in the deck.

5. **Presentation of a product**

   The moderator introduces a product, real or imaginary, for the players to role play. They can start with questions about the product with prompts such as:

   - Who will use the product?
   - What features will it have?
   - When will it be used?
   - Where will it be used?
   - Why is it useful?
   - How will it be used?

6. **Writing reviews**

   Players each write a review of the product or scenario based on the cards they've been dealt. Players take turns reading their reviews aloud. Listen carefully for any issues or themes that might come up. Take note of any changes you might make to your product based on insights discussed.

   - The stakeholder card tells them who they are for the round.

- The ethical principle card is one of the six Microsoft AI principles; it tells them what area they're focusing on in their review.
- The rating card tells them how much their stakeholder likes the technology (1, 3, or 5 stars).

The player then writes a review of the technology, from the perspective of their stakeholder, of why they like or dislike the product from the perspective of the ethical principle they received.

Discussion

The moderator has each player read their reviews. Everyone is invited to discuss the different perspectives and other considerations that may not have come up.

Potential moderator questions include:

- Who is the most impacted?
- What features are problematic?
- What are the potential harms?

Harms mitigation

Finally, the group picks a thread from the discussion to begin identifying design or process changes that can mitigate a particular harm. At the end of each round, the decks are shuffled, and another round can begin with the same or a different scenario.

Next steps

Once you have enough understanding of potential harms or negative impact your product or scenario may cause, proceed to learn how to model these harms so you can devise effective mitigations.

Foundations of assessing harm

11/19/2021 • 3 minutes to read • Edit Online

Harms Modeling is a practice designed to help you anticipate the potential for harm, identify gaps in product that could put people at risk, and ultimately create approaches that proactively address harm.

Why harms modeling?

As technology builders, your work is global. Designing AI to be trustworthy requires creating solutions that reflect ethical principles deeply rooted in important and timeless values. In the process of designing and building your technologies, it is essential to evaluate not only ideal outcomes, but possible negative ones as well.

Technology and human rights

It is as important as ever to be aware of how digital technology could impact human rights. In addition to continuing to protect privacy and security, we must address the risks of AI and other emerging technologies, such as facial recognition. History teaches us that human rights violations not only result from the nefarious use of technology, but also from a lack of awareness amongst those who have good intentions. As a part of our company's dedication to the protection of human rights, Microsoft forged a partnership with important stakeholders outside of our industry, including the United Nations (UN).

An important set of UN principles that our company firmly supports, which was ratified by over 250 nations, is the Universal Declaration of Human Rights (UDHR). The UDHR is a milestone document. Drafted by a diverse global group of legal and cultural experts, the Declaration was proclaimed by the United Nations General Assembly in 1948 as a common standard of achievements for all peoples and all nations. It sets out, for the first time, fundamental human rights to be universally protected. It has been translated into over 500 languages. Additionally, Microsoft is one of 4,700 corporate signatories to the UN Global Compact, an international business initiative designed to promote responsible corporate citizenship.

Download in Microsoft Word

Human understanding

In addition to appreciating the importance of human rights, building trustworthy systems requires us to consider many people's perspectives. Asking who the stakeholders are, what they value, how they could benefit, and how they could be hurt by our technology, is a powerful step that allows us to design and build better products.

Who does the technology impact?

Who are the customers?

- What do they value?
- How should they benefit?
- How could tech harm them?

Who are the non-customer stakeholders?

- What do they value?
- How should they benefit?
- How could tech harm them?

Asking these questions is a practice in Value Sensitive Design and is the beginning to better understanding what is important to stakeholders, and how it plays into their relationship with the product.

Types of stakeholders

Project sponsors

Backers, decision makers, and owners make up this category. Their values are articulated in the project strategy and goals.

Tech builders

Designers, developers, project managers, and people working directly on designing systems make up this group. They bring their own ethical standards and profession-specific values to the system.

Direct & indirect stakeholders

These stakeholders are significantly impacted by the system. Those who are impacted include end users, software staff, clients, bystanders, interfacing institutions, and even past or future generations. Non-human factors such as places, for example, historic buildings or sacred spaces, may also be included.

Marginalized populations

This category is made up of the population frequently considered a minority, vulnerable, or stigmatized. This category includes children, the elderly, members of the LGBTQ+ community, ethnic minorities, and other populations who often experience unique and disproportionate consequences.

Assessing Harm

Once you have defined the technology purpose, use cases, and stakeholders, conduct a Harms Modeling exercise to evaluate potential ways the use of a technology you are building could result in negative outcomes for people and society.

CATEGORY	TYPE OF HARM	CONTRIBUTING FACTORS	*Severity*	*Scale*	*Probability*	*Frequency*	POTENTIAL
Risk of injury	Physical or infrastructure damage			▼	▼	▼	LOW
	Emotional or psychological distress		▲	■	▲	▲	HIGH
Denial of consequential services	Opportunity loss			▼	▼		LOW
	Economic loss			▼	▼	▼	LOW
Infringement on human rights	Dignity loss		■	▼	■	▼	MODERATE
	Liberty loss		■	▼	▼	▼	LOW
	Privacy loss		▲	■	▲	▲	HIGH
	Environmental impact			▼	▼		LOW
Erosion of social & democratic structures	Manipulation		▲	■	■	▲	HIGH
	Social detriment		■	▼	■	▼	MODERATE

NOTE: This summary represents the outcome of a qualitative assessment and is used to inform prioritization of responsible innovation mitigations.

Download in Microsoft Word

The diagram above is an example of a harms evaluation. This model is a qualitative approach that's used to understand the potential magnitude of harm.

You can complete this ideation activity individually, but ideally it is conducted as collaboration between developers, data scientists, designers, user researcher, business decision-makers, and other disciplines that are involved in building the technology.

Suggestions for harm description statements:

- Intended use: If [feature] was used for [use case], then [stakeholder] could experience [harm description].
- Unintended use: If [user] tried to use [feature] for [use case], then [stakeholder] could experience [harm description].
- System error: If [feature] failed to function properly when used for [use case], then [stakeholder] could experience [harm description].
- Misuse: [Malicious actor] could potentially use [feature], to cause [harm description] to [stakeholder].

Use the categories, questions, and examples described in the Types of harm to generate specific ideas for how harm could occur. The article lists categories of harms, that are based upon common negative impact areas. Adapt and adopt additional categories that are relevant to you.

Next Steps

Read Types of harm for further harms analysis.

Types of harm

11/19/2021 • 13 minutes to read • Edit Online

This article creates awareness for the different types of harms, so that appropriate mitigation steps can be implemented.

Risk of injury

Physical injury

Consider how technology could hurt people or create dangerous environments.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Overreliance on safety features	This points out the dependence on technology to make decisions without adequate human oversight.	How might people rely on this technology to keep them safe? How could this technology reduce appropriate human oversight?	A healthcare agent could misdiagnose illness, leading to unnecessary treatment.
Inadequate fail-safes	Real-world testing may insufficiently consider a diverse set of users and scenarios.	If this technology fails or is misused, how would people be impacted? At what point could a human intervene? Are there alternative uses that have not been tested for? How would users be impacted by a system failure?	If an automatic door failed to detect a wheelchair during an emergency evacuation, a person could be trapped if there isn't an accessible override button.
Exposure to unhealthy agents	Manufacturing, as well as disposal of technology can jeopardize the health and well-being of workers and nearby inhabitants.	What negative outcomes could come from the manufacturing of your components or devices?	Inadequate safety measures could cause workers to be exposed to toxins during digital component manufacturing.

Emotional or psychological injury

Misused technology can lead to serious emotional and psychological distress.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Overreliance on automation	Misguided beliefs can lead users to trust the reliability of a digital agent over that of a human.	How could this technology reduce direct interpersonal feedback? How might this technology interface with trusted sources of information? How could sole dependence on an artificial agent impact a person?	A chat bot could be relied upon for relationship advice or mental health counseling instead of a trained professional.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Distortion of reality or gaslighting	When intentionally misused, technology can undermine trust and distort someone's sense of reality.	Could this be used to modify digital media or physical environments?	An IoT device could enable monitoring and controlling of an ex-intimate partner from afar.
Reduced self-esteem/reputation damage	Some shared content can be harmful, false, misleading, or denigrating.	How could this technology be used to inappropriately share personal information? How could it be manipulated to misuse information and misrepresent people?	Synthetic media "revenge porn" can swap faces, creating the illusion of a person participating in a video, who did not.
Addiction/attention hijacking	Technology could be designed for prolonged interaction, without regard for well-being.	In what ways might this technology reward or encourage continued interaction beyond delivering user value?	Variable drop rates in video game loot boxes could cause players to keep playing and neglect self-care.
Identity theft	Identity theft may lead to loss of control over personal credentials, reputation, and/or representation.	How might an individual be impersonated with this technology? How might this technology mistakenly recognize the wrong individual as an authentic user?	Synthetic voice font could mimic the sound of a person's voice and be used to access a bank account.
Misattribution	This includes crediting a person with an action or content that they are not responsible for.	In what ways might this technology attribute an action to an individual or group? How could someone be affected if an action was incorrectly attributed to them?	Facial recognition can misidentify an individual during a police investigation.

Denial of consequential services

Opportunity loss

Automated decisions could limit access to resources, services, and opportunities essential to well-being.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Employment discrimination	Some people may be denied access to apply for or secure a job based on characteristics unrelated to merit.	Are there ways in which this technology could impact recommendations or decisions related to employment?	Hiring AI could recommend fewer candidates with female-sounding names for interviews.
Housing discrimination	This includes denying people access to housing or the ability to apply for housing.	How could this technology impact recommendations or decisions related to housing?	Public housing queuing algorithm could cause people with international-sounding names to wait longer for vouchers.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Insurance and benefit discrimination	This includes denying people insurance, social assistance, or access to a medical trial due to biased standards.	Could this technology be used to determine access, cost, allocation of insurance or social benefits?	Insurance company might charge higher rates for drivers working night shifts due to algorithmic predictions suggesting increased drunk driving risk.
Educational discrimination	Access to education may be denied because of an unchangeable characteristic.	How might this technology be used to determine access, cost, accommodations, or other outcomes related to education?	Emotion classifier could incorrectly report that students of color are less engaged than their white counterparts, leading to lower grades.
Digital divide/technological discrimination	Disproportionate access to the benefits of technology, may leave some people less informed or less equipped to participate in society.	What prerequisite skills, equipment, or connectivity are necessary to get the most out of this technology? What might be the impact of select people gaining earlier access to this technology than others, in terms of equipment, connectivity, or other product functionality?	Content throttling could prevent rural students from accessing classroom instruction video feed.
Loss of choice/network and filter bubble	Presenting people with only information that conforms to and reinforces their own beliefs.	How might this technology affect which choices and information are made available to people? What past behaviors or preferences might this technology rely on to predict future behaviors or preferences?	News feed could only present information that confirms existing beliefs.

Economic loss

Automating decisions related to financial instruments, economic opportunity, and resources can amplify existing societal inequities and obstruct well-being.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Credit discrimination	People may be denied access to financial instruments based on characteristics unrelated to economic merit.	How might this technology rely on existing credit structures to make decisions? How might this technology affect the ability of an individual or group to obtain or maintain a credit score?	Higher introductory rate offers could be sent only to homes in lower socioeconomic postal codes.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Differential pricing of goods and services	Goods or services may be offered at different prices for reasons unrelated to the cost of production or delivery.	How could this technology be used to determine pricing of goods or services? What are the criteria for determining the cost to people for use of this tech?	More could be charged for products based on designation for men or women.
Economic exploitation	People might be compelled or misled to work on something that impacts their dignity or wellbeing.	What role did human labor play in producing training data for this technology? How was this workforce acquired? What role does human labor play in supporting this technology? Where is this workforce expected to come from?	Paying financially destitute people for their biometric data to train AI systems.
Devaluation of individual expertise	Technology may supplant the use of paid human expertise or labor.	How might this technology impact the need to employ an existing workforce?	AI agents replace doctors/radiographers for evaluation of medical imaging.

Infringement on human rights

Dignity loss

Technology can influence how people perceive the world, and how they recognize, engage, and value one another. The exchange of honor and respect between people can be interfered with.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Dehumanization	Removing, reducing, or obscuring visibility of a person's humanity.	How might this technology be used to simplify or abstract the way a person is represented? How might this technology reduce the distinction between humans and the digital world?	Entity recognition and virtual overlays in drone surveillance could reduce the perceived accountability of human actions.
Public shaming	This may mean exposing people's private, sensitive, or socially inappropriate material.	How might movements or actions be revealed through data aggregation?	A fitness app could reveal a user's GPS location on social media, indicating attendance at an Alcoholics Anonymous meeting.

Liberty loss

Automation of legal, judicial, and social systems can reinforce biases and lead to detrimental consequences.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Predictive policing	Inferring suspicious behavior and/or criminal intent based on historical records.	How could this support or replace human policing or criminal justice decision-making?	An algorithm can predict a number of area arrests, so police make sure they match, or exceed, that number.
Social control	Conformity may be reinforced or encouraged by publicly designating human behaviors as positive or negative.	What types of personal or behavioral data might feed this technology? How would it be obtained? What outputs would be derived from this data? Is this technology likely to be used to encourage or discourage certain behaviors?	Authoritarian government uses social media and e-commerce data to determine a "trustworthy" score based on where people shop and who they spend time with.
Loss of effective remedy	This means an inability to explain the rationale or lack of opportunity to contest a decision.	How might people understand the reasoning for decisions made by this technology? How might an individual that relies on this technology explain the decisions it makes? How could people contest or question a decision this technology makes?	Automated prison sentence or pre-trial release decision is not explained to the accused.

Privacy loss

Information generated by our use of technology can be used to determine facts or make assumptions about someone without their knowledge.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Interference with private life	Revealing information a person has not chosen to share.	How could this technology use information to infer portions of a private life? How could decisions based upon these inferences expose things that a person does not want made public?	Task-tracking AI could monitor personal patterns from which it infers an extramarital affair.
Forced association	Requiring participation in the use of technology or surveillance to take part in society.	How might use of this technology be required for participation in society or organization membership?	Biometric enrollment in a company's meeting room transcription AI is a stipulated requirement in job offer letter.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Inability to freely and fully develop personality	This may mean restriction of one's ability to truthfully express themselves or explore external avenues for self-development.	In what way does the system or product ascribe positive vs negative connotations toward particular personality traits? In what way can using the product or system reveal information to entities such as the government or employer that inhibits free expression?	Intelligent meeting system could record all discussions between colleagues including personal coaching and mentorship sessions.
Never forgiven	Digital files or records may never be deleted.	What and where is data being stored from this product, and who can access it? How long is user data stored after technology interaction? How is user data updated or deleted?	A teenager's social media history could remain searchable long after they have outgrown the platform.
Loss of freedom of movement or assembly	This means an inability to navigate the physical or virtual world with desired anonymity.	In what ways might this technology monitor people across physical and virtual space?	A real name could be required in order to sign up for a video game enabling real-world stalking.

Environmental impact

The environment can be impacted by every decision in a system or product life cycle, from the amount of cloud computing needed to retail packaging. Environmental changes can impact entire communities.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Exploitation or depletion of resources	Obtaining the raw materials for a technology, including how it's powered, leads to negative consequences to the environment and its inhabitants.	What materials are needed to build or run this technology? What energy requirements are needed to build or run this technology?	A local community could be displaced due to the harvesting of rare earth minerals and metals required for some electronic manufacturing.
Electronic waste	Reduced quality of collective well-being because of the inability to repair, recycle, or otherwise responsibly dispose of electronics.	How might this technology reduce electronic waste by recycling materials or allowing users to self-repair? How might this technology contribute to electronic waste when new versions are released or when current/past versions stop working?	Toxic materials inside discarded electronic devices could leach into the water supply, making local populations ill.

Erosion of social & democratic structures

Manipulation

The ability for technology to be used to create highly personalized and manipulative experiences can undermine an informed citizenry and trust in societal structures.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Misinformation	Disguising fake information as legitimate or credible information.	How might this technology be used to generate misinformation? How could it be used to spread misinformation that appears credible?	Generation of synthetic speech of a political leader sways an election.
Behavioral exploitation	This means exploiting personal preferences or patterns of behavior to induce a desired reaction.	How might this technology be used to observe patterns of behavior? How could this technology be used to encourage dysfunctional or maladaptive behaviors?	Monitoring shopping habits in connected retail environment leads to personalized incentives for impulse shoppers and hoarders.

Social detriment

At scale, the way technology impacts people shapes social and economic structures within communities. It can further ingrain elements that include or benefit some, at the exclusion of others.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Amplification of power inequality	This may perpetuate existing class or privilege disparities.	How might this technology be used in contexts where there are existing social, economic, or class disparities? How might people with more power or privilege disproportionately influence the technology?	Requiring a residential address & phone number to register on a job website could prevent a homeless person from applying for jobs.
Stereotype reinforcement	This may perpetuate uninformed "conventional wisdom" about historically or statistically underrepresented people.	How might this technology be used to reinforce or amplify existing social norms or cultural stereotypes? How might the data used by this technology cause it to reflect biases or stereotypes?	Results of an image search for "CEO" could primarily show photos of Caucasian men.
Loss of individuality	This may be an inability to express a unique perspective.	How might this technology amplify majority opinions or "group-think"? Conversely, how might unique forms of expression be suppressed. In what ways might the data gathered by this technology be used in feedback to people?	Limited customization options in designing a video game avatar inhibits self-expression of a player's diversity.

HARM	DESCRIPTION	CONSIDERATION(S)	EXAMPLE
Loss of representation	Broad categories of generalization obscure, diminish, or erase real identities.	How could this technology constrain identity options? Could it be used to automatically label or categorize people?	Automatic photo caption assigns incorrect gender identity and age to the subject.
Skill degradation and complacency	Overreliance on automation leads to atrophy of manual skills.	In what ways might this technology reduce the accessibility and ability to use manual controls?	Overreliance on automation could lead to an inability to gauge the airplane's true orientation because the pilots have been trained to rely on instruments only.

Evaluate harms

Once you have generated a broad list of potential harms, you should complete your Harms Model by evaluating the potential magnitude for each category of harm. This will allow you to prioritize your areas of focus. See the following example harms model for reference:

CONTRIBUTING FACTOR	DEFINITION
Severity	How acutely could an individual or group's well-being be impacted by the technology?
Scale	How broadly could the impact to well-being be experienced across populations or groups?
Probability	How likely is it that individual or group's well-being will be impacted by the technology?
Frequency	How often would an individual or group experience an impact to their well-being from the technology?

Next Steps

Use the Harms Model you developed to guide your product development work:

- Seek more information from stakeholders that you identified as potentially experiencing harm.
- Develop and validate hypothesis for addressing the areas you identified as having the highest potential for harm.
- Integrate the insights into your decisions throughout the technology development process: data collection and model training, system architecture, user experience design, product documentation, feedback loops, and communication capabilities and limitations of the technology.
- Explore Community Jury.
- Assess and mitigate unfairness using Azure Machine Learning and the open-source FairLearn package.

Other Responsible AI tools:

- Responsible AI resource center
- Guidelines for Human AI interaction
- Conversational AI guidelines

- Inclusive Design guidelines
- AI Fairness checklist

Additional references:

- Downloadable booklet for assessing harms
- Value Sensitive Design

Community jury

11/19/2021 • 7 minutes to read • Edit Online

Community jury, an adaptation of the citizen jury, is a technique where diverse stakeholders impacted by a technology are provided an opportunity to learn about a project, deliberate together, and give feedback on use cases and product design. This technique allows project teams to understand the perceptions and concerns of impacted stakeholders for effective collaboration.

A community jury is different from a focus group or market research; it allows the impacted stakeholders to hear directly from the subject matter experts in the product team, and cocreate collaborative solutions to challenging problems with them.

Benefits

This section discusses some important benefits of community jury.

Expert testimony

Members of a community jury hear details about the technology under consideration, directly from experts on the product team. These experts help highlight the capabilities in particular applications.

Proximity

A community jury allows decision-makers to hear directly from the community, and to learn about their values, concerns, and ideas regarding a particular issue or problem. It also provides a valuable opportunity to better understand the reasons for their conclusions.

Consensus

By bringing impacted individuals together and providing collaborative solutions and an opportunity for them to learn and discuss key aspects of the technology, a community jury can identify areas of agreement and build common ground solutions to challenging problems.

Community jury contributors

Product team

This group comprises owners who will bring the product to market, representative project managers, engineers, designers, data scientists, and others involved in the making of a product. Additionally, subject matter experts are included who can answer in-depth questions about the product.

Preparation

Effective deliberation and cocreation require ready-to-use answers to technical questions. As product owners, it is important to ensure technical details are collected prior to the start of the jury session.

Relevant artifacts could include:

- Documentation of existing protections, planned or in place.
- Data Flows, for example, what data types collected, who will have access, for how long, with what protections, and so on.
- Mockups or prototypes of proposed stakeholder collaborative solutions.

During the session

Along with providing an accessible presentation of technical details, product team witnesses should describe certain applications of the technology.

Relevant artifacts could include:

- Customer benefits
- Harms assessment and socio-technical impact
- Storyboards
- Competitive solutions and analyses
- Academic or popular media reports

Moderator

Bring on a neutral user researcher to ensure everyone is heard, avoiding domination of the communications by any one member. The moderator will facilitate brainstorms and deliberations, as well as educate jury members in uncovering bias, and ways to ask difficult questions. If a user researcher is not available, choose a moderator who is skilled at facilitating group discussions. Following the session, the moderator is responsible for the following:

- ensure that the agreed-upon compensation is provided to the jury members;
- produce a report that describes key insights, concerns, and recommendations, and
- share key insights and next steps with the jury, and thank them for their participation.

Preparation

- Structure the sessions so that there is ample time for learning, deliberation, and cocreation. This could mean having multiple sessions that go in-depth on different topics or having longer sessions.
- Pilot the jury with a smaller sample of community members to work out the procedural and content issues prior to the actual sessions.

During the session

- Ensure that all perspectives are heard, including those of the project team and those of the jury members. This minimizes group-thinking as well as one or two individuals dominating the discussion.
- Reinforce the value of the juror participation by communicating the plan for integrating jury feedback into the product planning process. Ensure that the project team is prepared to hear criticisms from the jury.

Jury members

These are direct and indirect stakeholders impacted by the technology, representative of the diverse community in which the technology will be deployed.

Sample size

A jury should be large enough to represent the diversity and collective will of the community, but not so large that the voices of quieter jurors are drowned out. It is ideal to get feedback from at least 16-20 individuals total who meet the criteria below. You can split the community jury over multiple sessions so that no more than 8-10 individuals are part of one session.

Recruitment criteria

Stratify a randomly selected participant pool so that the jury includes the demographic diversity of the community. Based on the project objectives, relevant recruitment criteria may include balancing across gender identity, age, privacy index. Recruitment should follow good user research recruitment procedures and reach a wider community.

Session structure

Sessions typically last 2-3 hours. Add more or longer deep dive sessions, as needed, if aspects of the project require more learning, deliberation, and cocreation.

1. **Overview, introduction, and Q&A:** The moderator provides a session overview, then introduces the project team and explains the product's purpose, along with potential use cases, benefits, and harms. Questions are then accepted from community members. This session should be between 30 to 60

minutes long.

2. **Discussion of key themes:** Jury members ask in-depth questions about aspects of the project, fielded by the moderator. This session should also be between 30 to 60 minutes in length.

3. This step can be any one of the following options:

   - **Deliberation and cocreation:** This option is preferable. Jury members deliberate and co-create effective collaboration solutions with the project team. This is typically 30 to 60 minutes long.
   - **Individual anonymous survey:** Alternatively, conduct an anonymous survey of jury members. Anonymity may allow issues to be raised that would not otherwise be expressed. Use this 30-minute session if there are time constraints.

4. **Following the session:** The moderator produces a study report that describes key insights, concerns, and potential solutions to the concerns.

If the values of different stakeholders were in conflict with each other during the session and the value tensions were left unresolved, the product team would need to brainstorm solutions, and conduct a follow-up session with the jury to determine if the solutions adequately resolve their concerns.

Tips to run a successful, effective, and collaborative jury

- Ensure alignment of goals and outcomes with the project team before planning begins, including deliverables, recruitment strategy, and timeline. Consider including additional subject-matter experts relevant to the project, to address open questions/concerns.
- The consent to audio and video recording of the jury should follow your company's standard procedures for non-disclosure and consent that is obtained from participants during user research studies.
- Provide fair compensation to participants for the time they devote to participation. Whenever possible, participants should also be reimbursed for costs incurred as a result of study participation, for example, parking and transportation costs. Experienced user research recruiters can help determine fair gratuities for various participant profiles.
- Ensure that all perspectives are heard, minimizing group-thinking as well as one or two individuals dominating the discussion. This should include those of the project team and the jury members.
- Structure the sessions so that there is ample time for learning, deliberation, and cocreation. This could mean having multiple sessions going in-depth on different topics or having longer sessions.
- Pilot the jury with a smaller sample of community members to work out the procedural and content issues prior to the actual sessions.
- If the topic being discussed is related to personal data privacy, aim to balance the composition of the community jury to include individuals with different privacy indices.
- Consider including session breaks and providing snacks/refreshments for sessions that are two hours or longer.
- Reinforce the value of the juror participation by communicating the plan for integrating jury feedback into the product planning process. Also ensure that the project team is prepared to hear criticisms from the jury.
- After the jury, in addition to publishing the research report, send out a thank you note to the volunteer participants of the jury, along with an executive summary of the key insights.

Additional information

Privacy index

The Privacy Index is an approximate measure for an individual's concern about personal data privacy, and is gauged using the following:

1. Consumers have lost all control over how personal information is collected and used by companies.
2. Most businesses handle the personal information they collect about consumers in a proper and confidential

way.

3. Existing laws and organizational practices provide a reasonable level of protection for consumer privacy today.

Participants are asked to provide responses to the above concerns using the scale of: 1 - Strongly Agree, 2 - Somewhat Agree, 3 - Somewhat Disagree, 4- Strongly Disagree, and classified into the categories below.

High/Fundamentalist => IF A = 1 or 2 AND B & C = 3 or 4

Low/unconcerned => IF A = 3 or 4 AND B & C = 1 or 2

Medium/pragmatist => All other responses

Next steps

Explore the following references for detailed information on this method:

- Jefferson center: creator of the Citizen's Jury method https://jefferson-center.org/about-us/how-we-work/
- Citizen's jury handbook http://www.rachel.org/files/document/Citizens_Jury_Handbook.pdf
- Case study: Connected Health Cities (UK)
  - Project page
  - Final report
  - Jury specification
  - Juror's report
- Case study: Community Jury at Microsoft

Architecture for startups

11/19/2021 • 5 minutes to read • Edit Online

Building a startup is a unique challenge. The core task is to find a place for an innovation as a product or service in the market. This process requires testing multiple assumptions that are built into the innovation. A successful startup must iterate through these assumptions, and grow and scale as its product gains product and market fit. After finding this fit, the startup must scale to meet market demands.

In the different startup life stages, developers, architects, and chief technical officers (CTOs) handle distinct phases of development. These stages require fundamentally different approaches and different technology choices. Part of the task is to establish which phase your startup is in. Choose the technologies and architectures that match that phase.

Innovation stages

Kent Beck describes a three-stage process of software product innovation. Those stages are *explore*, *expand*, and *extract*. You can think about the different parts of this process as a graph:

A graph showing a sigmoid curve plotted against a y-axis "Certainty/Investment/Risk of Change" and an x-axis "Time". The graph has three areas highlighted: the initial portion before upward inflection labeled "Explore", the high growth part of the sigmoid curve labeled "Expand" and the plateau labeled "Extract".

- The **Explore** stage starts with a low slope, where you're trying to find what works. Certainty is low, you only invest small amounts, and the risk from any changes you make is also low.
- At some point, the graph rises more rapidly. This rapid growth is the **Expand** stage. Your certainty greatly increases, you invest much more, and you're much more aware of risks.
- Finally, as the graph flattens out, you reach the **Extract** stage. The certainty, investment, and risk from change are all high, but the rate of growth has reached a plateau.

Explore

When your startup is in the exploration stage, your imperative is to invest small amounts of time and effort on many different product ideas. The fact that most ideas won't be right drives exploration. Only by iterating and learning can you find product and market fit. By making many small bets, you aim to find a product idea that pays off.

This stage requires discipline. It's easy to overinvest in an idea that you could test with less time and energy. A technologist finds it especially easy to fall into this trap. To make architectural choices that ease exploration, remember that you're exploring. You don't yet know if the current product idea is one that will scale.

From an architecture perspective, choose services that optimize for speed, cost, and options. Use managed services and platforms as a service (PaaS) like Azure App Service to get started quickly without worrying about complex infrastructure. Manage costs by choosing smaller instance sizes while you're exploring. Containers support developing with whatever tools make sense for you.

Build your first stack

As with your first product version, your first technology stack should be firmly rooted in exploration. That means the technology stack should ease rapid product iteration without wasting effort. You don't want to spend time or effort on infrastructure or architecture that isn't required for answering current questions.

During the exploration phase, you need to optimize for speed, cost, and optionality. Speed is about how fast you can build and move forward with an idea, or move onto the next idea. Cost is how much you're spending to run your infrastructure. Optionality describes how fast you can change directions given the current architecture.

It's important to balance cost, speed, and optionality. Too much focus on cost limits speed and optionality. Too much focus on speed can lead to increased costs and fewer options. Designing for too many options builds complexity, which increases costs and reduces speed.

Expand

Once your startup finds growth through exploration, you shift gears to expansion. You focus on removing any blockages to your product's and company's continued growth. From a technical perspective, you solve infrastructure scale challenges and increase development velocity. The goals are to meet your new customers' needs and advance your product roadmap.

Extend your architecture

As you iterate on your product, you'll inevitably find areas where your architecture needs extending. You might need to complete long-running tasks in the background, or handle frequent updates from internet-of-things (IoT) devices. You might need to add full-text search or artificial intelligence to your product.

You might need architectural changes to accommodate items on your roadmap. Resist the temptation to make those changes too far in advance. Extensions risk adding complexity to your architecture and infrastructure costs to your balance sheet.

In early startup stages, any architecture extension should be just-in-time. The extension should take only as much time and energy as needed to test the next hypothesis. Be ready to remove extensions to reduce complexity. Look for product features that your customers aren't using as opportunities to simplify your architecture and reduce your infrastructure spending.

Extract

In the extraction phase, the pace of growth slows as you reach the limits of the market opportunity. Because you expanded through the previous phase, there's now a lot to lose, so you take a more cautious approach. Margin expansion, cost reduction, and efficiency improvements characterize the extraction phase. During the extraction phase, be careful not to compromise the product for the customers you won in the expansion phase.

Handle growth and mature your stack

Once a product achieves product and market fit, many demands drive its architecture. Increased usage might require infrastructure scaling to handle load. New enterprise compliance requirements might require greater isolation. These changes are common steps in maturing a successful application.

Changes you make to handle growth and add maturity are different from extending architecture. These changes aren't functional requirements, but relate to unlocking scale. Increased scale can come from net new customers, increased usage from existing customers, and customers with higher regulatory requirements.

Resist the temptation to optimize prematurely. Make sure to take growth and maturation steps that can help you continue iterating and improving your product.

Next steps

- See and deploy an example Core startup stack architecture.

Related resources

- Best practices in cloud applications
- Ten design principles for Azure applications

Architectural considerations for a multitenant solution

11/19/2021 • 2 minutes to read • Edit Online

When you're considering a multitenant architecture, there are a number of decisions you need to make and elements you need to consider.

In a multitenant architecture, you share some or all of your resources between tenants. This means that a multitenant architecture can give you cost and operational efficiency. However, multitenancy introduces complexities, including the following:

- How do you define what a *tenant* is, for your specific solution? Does a tenant correspond to a customer, a user, or a group of users (like a team)?
- How will you deploy your infrastructure to support multitenancy, and how much isolation will you have between tenants?
- What commercial pricing models will your solution offer, and how will your pricing models affect your multitenancy requirements?
- What level of service do you need to provide to your tenants? Consider performance, resiliency, security, and compliance requirements, like data residency.
- How do you plan to grow your business or solution, and will it scale to the number of tenants you expect?
- Do any of your tenants have unusual or special requirements? For example, does your biggest customer need higher performance or stronger guarantees than others?
- How will you monitor, manage, automate, scale, and govern your Azure environment, and how will multitenancy impact this?

Whatever your architecture, it's essential that you have a clear understanding of your customers' or tenants' requirements. If you have made sales commitments to customers, or if you have contractual obligations or compliance requirements to meet, then you need to know what those requirements are, when you architect your solution. But equally, your customers may have implicit expectations about how things *should* work, or how you *should* behave, which could affect the way you design a multitenant solution.

As an example, imagine you're building a multitenant solution that you sell to businesses in the financial services industry. Your customers have very strict security requirements, and they need you to provide a comprehensive list of every domain name that your solution uses, so they can add it to their firewall's allowlist. This requirement affects the Azure services you use and the level of isolation that you have to provide between your tenants. They also require that their solution has a minimum level of resiliency. There may be many similar expectations, both explicit and implicit, that you need to consider across your whole solution.

In this series, we outline the considerations that you should give, the requirements you should elicit, and some of the tradeoffs you need to make, when you are planning a multitenant architecture.

Intended audience

The content in this series is particularly relevant for technical decision-makers, like chief technology officers (CTOs) and architects. However, anyone who works with multitenant architectures should have some familiarity with these principles and tradeoffs.

Next steps

Consider different tenancy models for your solution.

Tenancy models to consider for a multitenant solution

11/19/2021 • 12 minutes to read • Edit Online

There are many different ways that you can design a solution to be multitenanted. Mostly this decision hinges on whether and how you share resources between your tenants. Intuitively, you might want to avoid sharing *any* resources, but this quickly becomes expensive, as your business scales and as you onboard more and more tenants.

It's helpful to think about the different models of multitenancy, by first understanding how you define tenants for your specific organization, what business drivers you have, and how you plan to scale your solution.

Define a tenant

First, you need to define a *tenant* for your organization. Consider who your customer is. In other words, who are you providing your services to? There are two common models:

- **Business-to-business (B2B)**. If your customers are other organizations, you are likely to consider your tenants to be those customers. However, consider whether your customers might have divisions (teams or departments), or if they have a presence in multiple countries. You may need to consider having a single customer map to multiple tenants, if there are different requirements for these subgroups. Similarly, a customer might want to maintain two instances of your service, so they can keep their development and production environments separated from each other. Generally, a single tenant will have multiple users. For example, all of your customer's employees will be users within the same tenant.
- **Business-to-consumer (B2C)**. If your customers are consumers, it's often more complicated to relate customers, tenants, and users. In some scenarios, each consumer could be their own tenant. However, consider whether your solution might be used by families, groups of friends, clubs, associations, or other groupings that might need to access and manage their data together. For example, a music-streaming service might support both individual users and families, and it might treat each of these account types differently, when it comes to separating them into tenants.

Your definition of *tenant* will impact some of the things that you need to consider or emphasize, when you architect your solution. For example, consider these different types of tenants:

- If your tenants are individual people or families, you may need to be particularly concerned about how you handle personal data, and the data sovereignty laws within each jurisdiction you serve.
- If your tenants are businesses, you may need to be mindful of your customers' requirements for regulatory compliance, the isolation of their data, and ensuring you meet a specified service-level objective (SLO), such as uptime or service availability.

How do you decide which model to use?

Selecting a tenancy model isn't just a technical decision; it's also a commercial decision that you need to make. You need to consider questions like the following:

- What are your business objectives?
- Will your customers accept all forms of multitenancy? How would each multitenancy model impact your compliance requirements, or your customer's compliance requirements?
- Will a single-tenant solution scale to your future growth aspirations?
- How large is your operations team, and how much of your infrastructure management are you able to

automate?

- Do your customers expect you to meet service-level agreements (SLAs), or do you have SLOs that you are aiming for?

If you expect that your business is going to scale to a large number of customers, it will be very important to deploy shared infrastructure. Otherwise, you'll have to maintain a large and ever-growing fleet of instances. Deploying individual Azure resources, for each customer, is likely to be unsustainable, unless you provision and use a dedicated subscription, per tenant. When sharing the same Azure subscription across multiple tenants, Azure resource quotas and limits might start to apply, and the operational costs to deploy and reconfigure these resources become higher, with each new customer.

Conversely, if you expect that your business is only going to have a few customers, it might be worth considering to have single-tenant resources that are dedicated to each customer. Similarly, if your customers' isolation requirements are high, a single-tenant infrastructure might be appropriate.

Logical and physical tenants

Next, you need to determine what a tenant means for your particular solution, and whether you should distinguish between *logical* and *physical* tenants.

For example, consider a music streaming service. Initially, you might build a solution that can easily cope with thousands (or even tens of thousands) of users. As you continue to grow, though, you might find that you need to duplicate your solution or some of its components, in order to scale to your new customer demand. This means that you'll need to work out how to assign specific customers to specific instances of your solution. You might do this randomly, or geographically, or by filling up a single instance and then spilling over to another. However, you will probably need to maintain a record of which customers you have and which infrastructure their data and applications are available on, so that you can route their traffic to the correct infrastructure. In this example, you might represent each user as a separate logical tenant, and then map the users to physical tenants that represent the different instances you've deployed. You have a one-to-many mapping between logical and physical tenants, and you can move logical tenants between physical tenants at your own discretion.

In contrast, consider a company that builds cloud software for legal firms. Your customers might insist on their own dedicated infrastructure to maintain their compliance standards. Therefore, you need to be prepared to deploy and manage many different instances of your solution, right from the start. In this example, a logical and physical tenant is the same thing.

A key difference between logical and physical tenants is how isolation is enforced. When multiple logical tenants share a single set of infrastructure, you typically rely on your application code and a tenant identifier in a database, to keep each tenant's data separate. When you have physical tenants, they have their own infrastructure, and so it may be less important for your code to be aware that it's operating in a multitenant environment.

Sometimes, you'll see physical tenants referred to as *deployments*, *supertenants*, or *stamps*. In the rest of this document, we use the terms *deployments* and *physical deployments*, to avoid confusion between logical and physical tenants.

When you receive a request for a specific logical tenant, you need to map it to the physical deployment that holds that tenant's data, as illustrated below:

Tenant isolation

One of the biggest considerations when designing a multitenant architecture is the level of isolation that each tenant needs. Isolation can mean different things:

- Having a single set of shared infrastructure, with separate instances of your application and separate databases for each tenant.
- Sharing some common resources, while keeping other resources separate for each tenant.
- Keeping data on a separate physical infrastructure. In the cloud, this might require separate Azure resources for each tenant, or it could even mean literally deploying a separate physical infrastructure, by using dedicated hosts.

Rather than thinking of isolation as being a discrete property, you should think about isolation as being a continuum. You can deploy components of your architecture that are more or less isolated than other components in the same architecture, depending on your requirements. The following diagram demonstrates a continuum of isolation:

The level of isolation impacts many aspects of your architecture, including the following:

- **Security.** If you share infrastructure between multiple tenants, you need to be especially careful not to access data from one tenant when returning responses to another. You need a strong foundation for your identity strategy, and you need to consider both tenant and user identity within your authorization process.
- **Cost.** Shared infrastructure can be used by multiple tenants, so it's cheaper.
- **Performance.** If you're sharing infrastructure, your system's performance may suffer as more customers use it, since the resources may be consumed faster.
- **Reliability.** If you're using a single set of shared infrastructure, a problem with one tenant's components can result in an outage for everyone.

- **Responsiveness to individual tenants' needs.** When you deploy infrastructure that is dedicated to one tenant, you may be able to tune the configuration for the resources for that specific tenant's requirements. You might even consider this in your pricing model, where you enable customers to pay more for isolated deployments.

Your solution architecture can influence the options that you've got available to you for isolation. For example, let's think about an example three-tier solution architecture:

- Your user interface tier might be a shared multitenant web app, and all of your tenants access a single hostname.
- Your middle tier could be a shared application layer, with shared message queues.
- Your data tier could be isolated databases, tables, or blob containers.

You can consider mixing and matching different levels of isolation at each tier. Your decision about what is shared and what is isolated will be based on many considerations, including cost, complexity, your customers' requirements, and the number of resources that you can deploy before reaching Azure quotas and limits.

Common tenancy models

Once you've established your requirements, evaluate them against some common tenancy models and patterns.

Automated single-tenant deployments

In an automated single-tenant deployment model, you deploy a dedicated set of infrastructure for each tenant, as illustrated in this example:

Your application is responsible for initiating and coordinating the deployment of each tenant's resources. Typically, solutions built using this model make extensive use of infrastructure as code (IaC) or the Azure Resource Manager APIs. You might use this approach when you need to provision entirely separate infrastructures for each of your customers. Consider the Deployment Stamps pattern when planning your deployment.

Benefits: A key benefit of this approach is that data for each tenant is isolated, which reduces the risk of accidental leakage. This can be important to some customers with high regulatory compliance overhead. Additionally, tenants are unlikely to affect each other's system performance, which is sometimes called the *noisy neighbor* problem. Updates and changes can be rolled out progressively across tenants, which reduces the likelihood of a system-wide outage.

Risks: Your cost efficiency is low, because you aren't sharing infrastructure between your tenants. If a single tenant requires spending a certain amount on infrastructure, then it's likely that 100 tenants will require 100 times that cost, in expenditure. Additionally, ongoing maintenance (like applying new configuration or software updates) is likely to be time-consuming. Consider automating your operational processes, and consider applying changes progressively through your environments. You should also consider other cross-deployment operations, like reporting and analytics across your whole estate. Likewise, ensure you plan for how you can query and manipulate data across multiple deployments.

Fully multitenant deployments

At the opposite extreme, you can consider a fully multitenant deployment, where all components are shared. You only have one set of infrastructure to deploy and maintain, and all tenants use it, as illustrated in the following diagram:

Benefits: This model is attractive because of the lower cost to operate a solution with shared components. Even if you need to deploy higher tiers or SKUs of resources, it's still often the case that the overall deployment cost is lower than a set of single-tenant resources. Additionally, if a user or tenant needs to move their data into another logical tenant, you don't have to migrate data between two separate deployments.

Risks:

- Take care to ensure you separate data for each tenant, and do not leak data between tenants. You may need to manage sharding your data yourself. Additionally, you may need to be concerned about the effects that individual tenants can have on the overall system. For example, if a single large tenant tries to perform a heavy query or operation, will it affect other tenants?
- Determine how you track and associate your Azure costs to tenants, if this is important to you. Maintenance can be simpler with a single deployment, since you only have to update one set of resources. However, it's also often riskier, since any changes may affect your entire customer base.
- Scale can be a factor to consider as well. You are more likely to reach Azure resource scale limits when you have a shared set of infrastructure. For example, if you use a storage account as part of your solution, then as your scale increases, the number of requests to that storage account could reach the limit of what the storage account can handle. To avoid hitting a resource quota limit, you might consider deploying multiple instances of your resources (for example, multiple AKS clusters or storage accounts), or you might even consider distributing your tenants across resources that you've deployed into multiple Azure subscriptions.
- There is likely to be a limit to how far you can scale a single deployment, and the costs of doing so may increase non-linearly. For example, if you have a single, shared database, when you run at very high scale you may exhaust its throughput and have to pay increasingly more for increased throughput, to keep up with your demand.

Vertically partitioned deployments

You don't have to sit at the extremes of these scales. Instead, you could consider vertically partitioning your tenants, with the following steps:

- Use a combination of single-tenant and multitenant deployments. For example, you might have most of your customers' data and application tiers on multitenant infrastructures, but you might deploy single-tenant infrastructures for customers who require higher performance or data isolation.
- Deploy multiple instances of your solution geographically, and have each tenant pinned to a specific deployment. This is particularly effective when you have tenants in different geographies.

Here's an example that illustrates a shared deployment for some tenants, and a single-tenant deployment for another:

Benefits: Since you are still sharing infrastructure, you can still gain some of the cost benefits of having shared multitenant deployments. You can deploy cheaper, shared resources for certain customers, like those who are trying your service with a trial. You can even bill customers a higher rate to be on a single-tenant deployment, thereby recouping some of your costs.

Risks: Your codebase will likely need to be designed to support both multitenant and single-tenant deployments. If you plan to allow migration between infrastructures, you need to consider how you migrate customers from a multitenant deployment to their own single-tenant deployment. You also need to have a clear understanding of which of your logical tenants are on which sets of physical infrastructure, so that you can communicate information about system issues or upgrades to the relevant customers.

Horizontally partitioned deployments

You can also consider horizontally partitioning your deployments. This means you have some shared components, while maintaining other components with single-tenant deployments. For example, you could build a single application tier, and then deploy individual databases for each tenant, as shown in this illustration:

Benefits: Horizontally partitioned deployments can help you mitigate a noisy-neighbor problem, if you've identified that most of the load on your system is due to specific components that you can deploy separately for each tenant. For example, your databases might absorb most of your system's load, because the query load is high. If a single tenant sends a large number of requests to your solution, the performance of a database might be negatively affected, but other tenants' databases (and shared components, like the application tier) remain unaffected.

Risks: With a horizontally partitioned deployment, you still need to consider the automated deployment and management of your components, especially the components used by a single tenant.

Next steps

Consider the lifecycle of your tenants.

Tenant lifecycle considerations in a multitenant solution

11/19/2021 • 4 minutes to read • Edit Online

When you're considering a multitenant architecture, it's important to consider all of the different stages in a tenant's lifecycle.

Trial tenants

For SaaS solutions, consider that many customers request or require trials, before they commit to purchase a solution. Trials bring along the following unique considerations:

- Should the trial data be subject to the same data security, performance, and service-level requirements as the data for full customers?
- Should you use the same infrastructure for trial tenants as for full customers, or should you have dedicated infrastructure for trial tenants?
- If customers purchase your service after a trial, how will they migrate the data from their trial tenants into their paid tenants?
- Are there limits around who can request a trial? How can you prevent abuse of your solution?
- What limits do you want or need to place on trial customers, such as time limits, feature restrictions, or limitations around performance?

Onboard new tenants

When onboarding a new tenant, consider the following:

- Will onboarding be a self-service, automated, or manual process?
- Does the customer have any specific requirements for data residency? For example, are there data sovereignty regulations in effect?
- Does the customer have to meet any compliance standards (such as PCI DSS, HIPAA, and so on)?
- Does the customer have any specific disaster recovery requirements, such as a recovery time objective (RTO) or a recovery point objective (RPO)? Are these different from the guarantees that you provide to other customers?
- What information do you require, to be able to fully onboard the customer?
- Does the platform provide different pricing options and billing models?
- Does the customer require pre-production environments? And are there set expectations on availability for that environment? Is it transient (on-demand) or always available?

Once tenants have been onboarded, they move into a 'business as usual' mode. However, there are still several important lifecycle events that can occur, even when they are in this mode.

Update tenants' infrastructure

You will need to consider how you apply updates to your tenants' infrastructures. Different tenants may have updates applied at different times. See Updates for other considerations about updating tenants' deployments.

Scale tenants' infrastructure

Consider whether your tenants might have seasonal business patterns, or otherwise change the level of consumption for your solution. For example, if you provide a solution to retailers, you expect that certain times of the year will be particularly busy in some geographic regions, and quiet at other times. Consider whether this affects the way you design and scale your solution, and be aware of *noisy neighbor* issues, when a subset of tenants scales unexpectedly and impacts the performance to other tenants. You can consider applying mitigations, which might include scaling individual tenants' infrastructures, moving tenants between deployments, and provisioning a sufficient level of capacity to handle spikes and troughs in traffic.

Move tenants between infrastructures

You might need to move tenants between infrastructures for a number of reasons, including the following:

- You are vertically partitioning your customers and choose to rebalance your tenants across your infrastructures or deployments.
- Customers are upgrading a SKU or pricing tier, and they need to be moved to a single-tenant, dedicated deployment with higher isolation from other tenants.
- Customers request their data to be moved to a dedicated data store.
- Customers require their data be moved to a new geographic region. This might occur during company acquisitions, or when laws or geopolitical situations change.

Consider how you move your tenants' data, as well as redirect requests to the new set of infrastructure that hosts their instance. You should also consider whether moving a tenant will result in downtime, and make sure tenants are fully aware of this.

Merge and split tenants

It's tempting to think of tenants or customers as static, unchanging entities. However, in reality, this often isn't true. For example:

- In business scenarios, companies might be acquired or merge, including companies located in different geographic regions.
- Similarly, in business scenarios, companies might split or divest.
- In consumer scenarios, individual users might join or leave families.

Consider whether you need to provide capabilities to manage the merging and separation of data, user identities, and resources. Also, consider how data ownership affects your handling of merge and split operations. For example, consider a consumer photography application built for families to share photos with one another. Are the photos owned by the individual family members who contributed them, or by the family as a whole? If users leave the family, should their data be removed or remain in the family's data set? If users join another family, should their old photos move with them?

Offboard tenants

It's also inevitable that tenants will occasionally need be removed from your solution. In a multitenant solution, this brings along some important considerations, including the following:

- How long should you maintain the customer data? Are there legal requirements to destroy data, after a certain period of time?
- Should you provide the ability for customers to be re-onboarded?
- If you run shared infrastructure, do you need to rebalance the allocation of tenants to infrastructure?

Deactivate and reactivate tenants

There are situations where a customer's account might need to be deactivated or reactivated. For example:

- The customer has requested deactivation. In a consumer system, a customer might opt to unsubscribe.
- The customer can't be billed, and you need to deactivate the subscription.

Deactivation is separate to offboarding in that it's intended to be a temporary state. However, after a period of time, you might choose to offboard a deactivated tenant.

Next steps

Consider the pricing models you will use for your solution.

Pricing models for a multitenant solution

11/19/2021 • 19 minutes to read • Edit Online

A good pricing model ensures that you remain profitable as the number of tenants grows and as you add new features. An important consideration when developing a commercial multitenant solution is how to design pricing models for your product.

When you determine the pricing model for your product, you need to balance the *return on value* (ROV) for your customers with the *cost of goods sold* (COGS) to deliver the service. Offering more flexible commercial models (for a solution) might increase the ROV for customers, but it might also increase the architectural and commercial complexity of the solution (and therefore also increase your COGS).

Some important considerations that you should take into account, when developing pricing models for a solution, are as follows:

- Will the COGS be higher than the profit you earn from the solution?
- Can the COGS change over time, based on growth in users or changes in usage patterns?
- How difficult is it to measure and record the information required to operate the pricing model? For example, if you plan to bill your customers based on the number of API calls they make, have you identified how you'll measure the API calls made by each customer?
- Does your profitability depend on ensuring customers use your solution in a limited way?
- If a customer overuses the solution, does that mean you're no longer profitable?

There are some key factors that influence your profitability:

- **Azure service pricing models.** The pricing models of the Azure or third-party services that make up your solution may affect which models will be profitable.
- **Service usage patterns.** Users may only need to access your solution during their working hours or may only have a small percentage of high-volume users. Can you reduce your COGS by reducing the unused capacity when your usage is low?
- **Storage growth.** Most solutions accumulate data over time. More data means a higher cost to store and protect it, reducing your profitability per tenant. Can you set storage quotas or enforce a data retention period?
- **Tenant isolation.** The tenancy model you use affects the level of isolation you have between your tenants. If you share your resources, do you need to consider how tenants might over-utilize or abuse the service? How will this affect your COGS and performance for everyone? Some pricing models are not profitable without additional controls around resource allocation. For example, you might need to implement service throttling to make a flat-rate pricing model sustainable.
- **Tenant lifecycle.** For example, solutions with high customer churn rates, or services that require a greater on-boarding effort, may suffer lower profitability--especially if they are priced using a consumption-based model.
- **Service level requirements.** Tenants that require higher levels of service may mean your solution isn't profitable anymore. It's critical that you're clear about your customers' service-level expectations and any obligations you have, so that you can plan your pricing models accordingly.

Common pricing models

There are a number of common pricing models that are often used with multitenant solutions. Each of these pricing models has associated commercial benefits and risks, and requires additional architectural considerations. It's important to understand the differences between these pricing models, so that you can

ensure your solution remains profitable as it evolves.

> **NOTE**
> You can offer multiple models for a solution or combine models together. For example, you could provide a per-user model for your customers that have fairly stable user numbers, and you can also offer a consumption model for customers who have fluctuating usage patterns.

Consumption-based pricing

A consumption model is sometimes referred to as *pay-as-you-go*, or *PAYG*. As the use of your service increases, your revenue increases:

When you measure consumption, you can consider simple factors, such as the amount of data being added to the solution. Alternatively, you might consider a combination of usage attributes together. Consumption models offer a number of benefits, but they can be difficult to implement in a multitenant solution.

Benefits: From your customers' perspective, there is minimal upfront investment that's required to use your solution, so that this model has a low barrier to entry. From your perspective as the service operator, your hosting and management costs increase as your customers' usage and your revenue increases. This increase can make it a highly scalable pricing model. Consumption pricing models work especially well when the Azure services that are used in the solution are consumption-based too.

Complexity and operational cost: Consumption models rely on accurate measurements of usage and on splitting this usage by tenant. This can be challenging, especially in a solution with many distributed components. You need to keep detailed consumption records for billing and auditing.

Risks: Consumption pricing can motivate your customers to reduce their usage of your system, in order to reduce their costs. Additionally, consumption models result in unpredictable revenue streams. You can mitigate this by offering *capacity reservations*, where customers pay for some level of consumption upfront. You, as the service provider, can use this revenue to invest in improvements in the solution, to reduce the operational cost or to increase the return on value by adding features.

> **NOTE**
> Implementing and supporting capacity reservations may increase the complexity of the billing processes within your application. You might also need to consider how customers get refunds or exchange their capacity reservations, and these processes can also add commercial and operational challenges.

Per-user pricing

A per-user pricing model involves charging your customers based on the number of people using your service, as demonstrated in the following diagram.

Per-user pricing models are very common, due to their simplicity to implement in a multitenant solution. However, they are associated with several commercial risks.

Benefits: When you bill your customers for each user, it's easy to calculate and forecast your revenue stream. Additionally, assuming that you have fairly consistent usage patterns for each user, then revenue increases at the same rate as service adoption, making this a scalable model.

Complexity and operational cost: Per-user models tend to be easy to implement. However, in some situations, you need to measure per-user consumption, which can help you to ensure that the COGS for a single user remains profitable. By measuring the consumption and associating it with a particular user, you can increase the operational complexity of your solution.

Risks: Different user consumption patterns might result in a reduced profitability. For example, heavy users of the solution might cost more to serve than light users. Additionally, the actual return on value (ROV) for the solution is not reflected by the actual number of user licenses purchased.

Per-active user pricing

This model is similar to per-user pricing, but rather than requiring an upfront commitment from the customer on the number of expected users, the customer is only charged for users that actually log into and use the solution over a period (as shown in the following diagram).

You can measure this in whatever period makes sense. Monthly periods are common, and then this metric is often recorded as *monthly active users* or *MAU*.

Benefits: From your customers' perspective, this model requires a low investment and risk, because there is minimal waste; unused licenses aren't billable. This makes it particularly attractive when marketing the solution or growing the solution for larger enterprise customers. From your perspective as the service owner, your ROV is more accurately reflected to the customer by the number of monthly active users.

Complexity and operational cost: Per-active user models require you to record actual usage, and to make it available to a customer as part of the bill. Measuring per-user consumption helps to ensure profitability is maintained with the COGS for a single user, but again it requires additional work to measure the consumption for each user.

Risks: Like per-user pricing, there is a risk that the different consumption patterns of individual users may affect your profitability. Compared to per-user pricing, per-active user models have a less predictable revenue stream. Additionally, discount pricing doesn't provide a useful way of stimulating growth.

Per-unit pricing

In many systems, the number of users isn't the element that has the greatest effect on the overall COGS. For example, in device-oriented solutions, also referred to as the *internet of things* or *IoT*, the number of devices often has the greatest impact on COGS. In these systems, a per-unit pricing model can be used, where you define what a *unit* is, such as a device. See the following diagram.

Additionally, some solutions have highly variable usage patterns, where a small number of users has a disproportionate impact on the COGS. For example, in a solution sold to brick-and-mortar retailers, a per-store pricing model might be appropriate.

Benefits: In systems where individual users don't have a significant effect on COGS, per-unit pricing is a better way to represent the reality of how the system scales and the resulting impact to COGS. It also can improve the alignment to the actual patterns of usage for a customer. For many IoT solutions, where each device generates a predictable and constant amount of consumption, this can be an effective model to scale your solution's growth.

Complexity and operational cost: Generally, per-unit pricing is easy to implement and has a fairly low operational cost. However, the operational cost can become higher if you need to differentiate and track usage by individual units, such as devices or retail stores. Measuring per-unit consumption helps you ensure your profitability is maintained, since you can determine the COGS for a single unit.

Risks: The risks of a per-unit pricing model are similar to per-user pricing. Different consumption patterns by some units may mean that you have reduced profitability, such as if some devices or stores are much heavier users of the solution than others.

Feature- and service-level based pricing

You may choose to offer your solution with different tiers of functionality at different price points. For example, you might provide two monthly flat-rate or per-unit prices, one being a basic offering with a subset of features available, and the other presenting the comprehensive set of your solution's features. See the following diagram.

This model may also offer different service-level agreements for different tiers. For example, your basic tier may offer 99.9% uptime, whereas a premium tier may offer 99.99%. The higher service-level agreement (SLA) could be implemented by using services and features that enable higher availability targets.

Although this model can be commercially beneficial, it does require mature engineering practices to do well. With careful consideration, this model can be very effective.

Benefits: Feature-based pricing is often attractive to customers, since they can select a tier based on the feature set or service level they need. It also provides you with a clear path to upsell your customers with new features or higher resiliency for those who require it.

Complexity and operational cost: Feature-based pricing models can be complex to implement, since they require your solution to be aware of the features that are available at each price tier. Feature toggles can be an effective way to provide access to certain subsets of functionality, but this requires ongoing maintenance. Also, toggles increase the overhead to ensure high quality, because there will be more code paths to test. Enabling higher service availability targets in some tiers may require additional architectural complexity, to ensure the right set of infrastructure is used for each tier, and this process may increase the operational cost of the solution.

Risks: Feature-based pricing models can become complicated and confusing, if there are too many tiers or options. Additionally, the overhead involved in dynamically toggling features can slow down the rate at which you deliver additional features.

Freemium pricing

You might choose to offer a free tier of your service, with basic functionality and no service-level guarantees. You then might offer a separate paid tier, with additional features and a formal service-level agreement (as shown in the following diagram).

The free tier may also be offered as a time-limited trial, and during the trial your customers might have full or limited functionality available. This is referred to as a freemium model, which is effectively an extension of the feature-based pricing model.

Benefits: It's very easy to market a solution when it's free.

Complexity and operational cost: All of the complexity and operational cost concerns apply from the feature-based pricing model. However, you also have to consider the operational cost involved in managing free tenants. You might need to ensure that stale tenants are offboarded or removed, and you must have a clear retention policy, especially for free tenants. When promoting a tenant to a paid tier, you might need to move the tenant between Azure services, to obtain higher SLAs.

Risks: You need to ensure that you provide a high enough ROV for tenants to consider switching to a paid tier. Additionally, the cost of providing your solution to customers on the free tier needs to be covered by the profit margin from those who are on paid tiers.

Flat-rate pricing

In this model, you charge a flat rate to a tenant for access to your solution, for a given period of time. The same pricing applies regardless of how much they use the service, the number of users, the number of devices they connect, or any other metric. See the following diagram.

This is the simplest model to implement and for customers to understand, and it's often requested by enterprise customers. However, it can easily become unprofitable if you need to continue to add new features or if tenant consumption increases without any additional revenue.

Benefits: Flat-rate pricing is easy to sell, and it's easy for your customers to understand and budget for.

Complexity and operational cost: Flat-rate pricing models can be easy to implement because billing customers doesn't require any metering or tracking consumption. However, while not essential, it's advisable to measure per-tenant consumption to ensure that you're measuring COGS accurately and to ensure that your profitability is maintained.

Risks: If you have tenants who make heavy use of your solution, then it's easy for this pricing model to become unprofitable.

Discount pricing

Once you've defined your pricing model, you might choose to implement commercial strategies to incentivize growth through discount pricing. *Discount pricing* can be used with consumption, per-user, and per-unit pricing models.

> **NOTE**
> Discount pricing doesn't typically require architectural considerations, beyond adding support for a more complex billing structure. A complete discussion into the commercial benefits of discounting is beyond the scope of this document.

Common discount pricing patterns include:

- **Fixed pricing.** You have the same cost for each user, unit, or amount of consumption, no matter how much is purchased or consumed. This is the simplest approach. However, customers who make heavy use of your solution may feel like they should benefit from economies of scale by getting a discount.
- **Digressive pricing.** As customers purchase or consume more units, you reduce the cost per unit. This is more commercially attractive to customers.
- **Step pricing.** You reduce the cost per unit, as customers purchase more. However, you do so in step changes, based on predefined ranges of quantity. For example, you might charge a higher price for the first 100 users, then a lower price for the 101st to 200th user, then a lower price again after that. This can be more profitable.

The following diagram illustrates these pricing patterns.

Non-production environment discounts

In many cases, customers require access to a non-production environment that they can use for testing, training, or for creating their own internal documentation. Non-production environments usually have lower consumption requirements and costs to operate. For example, non-production environments often aren't subject to service-level agreements (SLAs), and rate limits might be set at lower values. You might also consider more aggressive autoscaling on your Azure services.

Equally, customers often expect non-production environments to be significantly cheaper than their production environments. There are several alternatives that might be appropriate, when you provide non-production environments:

- Offer a freemium tier, like you might already do for paid customers. This should be carefully monitored, as some organizations might create many testing and training environments, which will consume additional resources to operate.

  > **NOTE**
  > Time-limited trials using freemium tiers aren't usually suitable for testing and training environments. Customers usually need their non-production environments to be available for the lifetime of the production service.

- Offer a testing or training tier of your service, with lower usage limits. You may choose to restrict the availability of this tier to customers who have an existing paid tenant.
- Offer a discounted per-user, per-active user, or per-unit pricing for non-production tenants, with a lower or no service level agreement.
- For tenants using flat-rate pricing, non-production environments might be negotiated as part of the agreement.

> **NOTE**
> Feature-based pricing is not usually a good option for non-production environments, unless the features offered are the same as what the production environment offers. This is because tenants will usually want to test and provide training on all the same features that are available to them in production.

Unprofitable pricing models

An unprofitable pricing model costs you more to deliver the service than the revenue you earn. For example, you might charge a flat rate per tenant without any limits for your service, but then you would build your service with consumption-based Azure resources and without per-tenant usage limits. In this situation, you may

be at risk of your tenants overusing your service and, thus, making it unprofitable to serve them.

Generally, you want to avoid unprofitable pricing models. However, there are situations where you might choose to adopt an unprofitable pricing model, including:

- A free service is being offered to enable growth.
- Additional revenue streams are provided by services or add-on features.
- Hosting a specific tenant provides another commercial value, such as using them as an anchor tenant in a new market.

In the case that you inadvertently create an unprofitable pricing model, there are some ways to mitigate the risks associated with them, including:

- Limit the use of the service through usage limits.
- Require the use of capacity reservations.
- Request the tenant move to a higher feature or service tier.

Risky pricing models

When implementing a pricing model for a solution, you will usually have to make assumptions about how it will be used. If these assumptions turn out to be incorrect or the usage patterns change over time, then your pricing model may become unprofitable. Pricing models that are at risk of becoming unprofitable are known as *risky pricing* models. For example, if you adopt a pricing model that expects users to self-limit the amount that they use your solution, then you may have implemented a risky pricing model.

Most SaaS solutions will add new features regularly. This increases the ROV to users, which may also lead to increases in the amount the solution is used. This could result in a solution that becomes unprofitable, if the use of new features drives usage, but the pricing model doesn't factor this in.

For example, if you introduce a new video upload feature to your solution, which uses a consumption-based resource, and user uptake of the feature is higher than expected, then consider a combination of usage limits and feature and service level pricing.

Usage limits

Usage limits enable you to restrict the usage of your service in order to prevent your pricing models from becoming unprofitable, or to prevent a single tenant from consuming a disproportionate amount of the capacity of your service. This can be especially important in multitenant services, where a single tenant can impact the experience of other tenants by over-consuming resources.

> **NOTE**
>
> It's important to make your customers aware that you apply usage limits. If you implement usage limits without making your customers aware of the limit, then it will result in customer dissatisfaction. This means that it's important to identify and plan usage limits ahead of time. The goal should be to plan for the limit, and to then communicate the limits to customers before they become necessary.

Usage limits are often used in combination with feature and service-level pricing, to provide a higher amount of usage at higher tiers. Limits are also commonly used to protect core components that will cause system bottlenecks or performance issues, if they are over-consumed.

Rate limits

A common way to apply a usage limit is to add rate limits to APIs or to specific application functions. This is also referred to as throttling. Rate limits prevent continuous overuse. They are often used to limit the number of calls to an API, over a defined time period. For example, an API may only be called 20 times per minute, and it will return an HTTP 429 error, if it is called more frequently than this.

Some situations, where rate limiting is often used, include the following:

- REST APIs.
- Asynchronous tasks.
- Tasks that are not time sensitive.
- Actions that incur a high cost to execute.
- Report generation.

Implementing rate limiting can increase the solution complexity, but services like Azure API Management can make this simpler by applying rate limit policies.

Pricing model lifecycle

Like any other part of your solution, pricing models have a lifecycle. As your application evolves over time, you might need to change your pricing models. This may be driven by changing customer needs, commercial requirements, or changes to functionality within your application. Some common pricing lifecycle changes include the following:

- Adding a completely new pricing model. For example, adding a consumption pricing model to a solution that currently offers a flat rate model.
- Retiring an existing pricing model.
- Adding a tier to an existing pricing model.
- Retiring a tier in an existing pricing model.
- Changing usage limits on a feature in a pricing model.
- Adding or removing features from a feature and service-level pricing model.
- Changing from a business-to-consumer (B2C) commercial model, to a business-to-business (B2B) commercial model. This change then necessitates new pricing structures for your business customers.

It is usually complex to implement and manage many different pricing models at once. It's also confusing to your customers. So, it's better to implement only one or two models, with a small number of tiers. This makes your solution more accessible and easier to manage.

> **NOTE**
> Pricing models and billing functions should be tested, ideally using automated testing, just like any other part of your system. The more complex the pricing models, the more testing is required, and so the cost of development and new features will increase.

When changing pricing models, you will need to consider the following factors:

- Will tenants want to migrate to the new model?
- Is it easy for tenants to migrate to the new model?
- Will new pricing models expose your service to risk? For example, are you removing rate limits that are currently protecting critical resources from overuse?
- Do tenants have a clear path for migrating to the new pricing model?
- How will you prevent tenants from using older pricing models, so that you can retire them?
- Are tenants likely to receive *bill shock* (a negative reaction to an unexpected bill) upon changing pricing models?
- Are you monitoring the performance and utilization of your services, for new or changed pricing models, so that you can ensure continued profitability?
- Are you able to clearly communicate the ROV for new pricing models, to your existing tenants?

Next steps

Consider how you'll measure consumption by tenants in your solution.

Measure the consumption of each tenant

11/19/2021 • 6 minutes to read • Edit Online

As a solution provider, it's important to measure the consumption of each tenant in your multitenant solution. By measuring the consumption of each tenant, you can ensure that the cost of goods sold (COGS), for delivering the service to each tenant, is profitable.

There are two primary concerns driving the need for measuring each tenant's consumption:

- You need to measure the actual cost to serve each tenant. This is important to monitor the profitability of the solution for each tenant.
- You need to determine the amount to charge the tenant, when you're using consumption-based pricing.

However, it can be difficult to measure the actual resources used by a tenant in a multitenant solution. Most services that can be used as part of a multitenant solution don't automatically differentiate or break down usage, based on whatever you define a tenant to be. For example, consider a service that stores data for all of your tenants in a single relational database. It's difficult to determine exactly how much space each tenant uses of that relational database, either in terms of storage or of the compute capacity that's required to service any queries and requests.

By contrast, for a single-tenant solution, you can use Azure Cost Management within the Azure portal, to get a complete cost breakdown for all the Azure resources that are consumed by that tenant.

Therefore, when facing these challenges, it is important to consider how to measure consumption.

> **NOTE**
>
> In some cases, it's commercially acceptable to take a loss on delivering service to a tenant, for example, when you enter a new market or region. This is a commercial choice. Even in these situations, it's still a good idea to measure the consumption of each tenant, so that you can plan for the future.

Indicative consumption metrics

Modern applications (built for the cloud) are usually made up of many different services, each with different measures of consumption. For example, a storage account measures consumption based on the amount of data stored, the data transmitted, and the numbers of transactions. However, Azure App Service consumption is measured by the amount of compute resources allocated over time. If you have a solution that includes a storage account and App Service resources, then combining all these measurements together to calculate the actual COGS (cost of goods sold) can be a very difficult task. Often, it is easier to use a single indicative measurement to represent consumption in the solution.

For example, in the case of a multitenant solution that shares a single relational database, you might determine that the data stored is a good indicative consumption metric.

> **NOTE**
>
> Even if you use the volume of data stored by a tenant as an indicative consumption measure, it might not be a true representation of consumption for every tenant. For example, if a particular tenant does a lot more reads or runs more reporting from the solution, but it doesn't write a lot of data, then it could use a lot more compute than the storage requirements would indicate.

It is important to occasionally measure and review the actual consumption across your tenants, to determine whether the assumptions you're making about your indicative metrics are correct.

Transaction metrics

An alternative way of measuring consumption is to identify a key transaction that is representative of the COGS for the solution. For example, in a document management solution, it could be the number of documents created. This can be useful, if there is a core function or feature within a system that is transactional, and if it can be easily measured.

This method is usually easy and cost effective to implement, as there is usually only a single point in your application that needs to record the number of transactions that occur.

Per-request metrics

In solutions that are primarily API-based, it might make sense to use a consumption metric that is based around the number of API requests being made to the solution. This can often be quite simple to implement, but it does require you to use APIs as the primary interface to the system. There will be an increased operational cost of implementing a per-request metric, especially for high volume services, because of the need to record the request utilization (for audit and billing purposes).

> **NOTE**
> User-facing solutions that consist of a single-page application (SPA), or a mobile application that uses the APIs, may not be a good fit for the per-request metric. This is because there is little understanding by the end user of the relationship between the use of the application and the consumption of APIs. Your application might be chatty (it makes many API requests) or chunky (it makes too few API requests), and the user wouldn't notice a difference.

> **WARNING**
> Make sure you store request metrics in a reliable data store that's designed for this purpose. For example, although Azure Application Insights can track requests and can even track tenant IDs (by using properties), Application Insights is not designed to store every piece of telemetry. It removes data, as part of its sampling behavior. For billing and metering purposes, choose a data store that will give you a high level of accuracy.

Estimate consumption

When measuring the consumption for a tenant, it may be easier to provide an estimate of the consumption for the tenant, rather than trying to calculate the exact amount of consumption. For example, for a multitenant solution that serves many thousands of tenants in a single deployment, it is reasonable to approximate that the cost of serving the tenant is just a percentage of the cost of the Azure resources.

You might consider estimating the COGS for a tenant, in the following cases:

- You aren't using consumption-based pricing.
- The usage patterns and cost for every tenant is similar, regardless of size.
- Each tenant consumes a low percentage (say, <2%), of the overall resources in the deployment.
- The per-tenant cost is low.

You might also choose to estimate consumption in combination with indicative consumption metrics, transaction metrics, or per-request metrics. For example, for an application that primarily manages documents, the percentage of overall storage used by a tenant, to store its documents, gives a close enough representation of the actual COGS. This can be a useful approach, when measuring the COGS is difficult or when it would add

too much complexity to the application.

On-charging your costs

In some solutions, you can simply charge your customers the COGS for their tenant's resources. For example, you might use Azure resource tags to allocate billable Azure resources to tenants. You can then determine the cost to each tenant for the set of resources that's dedicated to them, plus a margin for profit and operation.

> **NOTE**
>
> Some Azure services don't support tags. For these services, you will need to attribute the costs to a tenant, based on the resource name, resource group, or subscription.

Azure Cost Analysis can be used to analyze Azure resource costs for single tenant solutions that use tags, resource groups, or subscriptions to attribute costs.

However, this becomes prohibitively complex in most modern multitenant solutions, because of the challenge of accurately determining the exact COGS to serve a single tenant. This method should only be considered for very simple solutions, solutions that have single-tenant resource deployments, or custom tenant-specific add-on features within a larger solution.

Some Azure services provide features that allow other methods of attribution of costs in a multitenant environment. For example, Azure Kubernetes Service supports multiple node pools, where each tenant is allocated a node pool with node pool tags, which are used to attribute costs.

Next steps

Consider the update deployment model you will use.

Considerations for updating a multitenant solution

11/19/2021 • 7 minutes to read • Edit Online

One of the benefits of cloud technology is continuous improvement and evolution. As a service provider, you need to apply updates to your solution: you might need to make changes to your Azure infrastructure, your code/applications, your database schemas, or any other component. It's important to plan how you update your environments. In a multitenant solution, it's particularly important to be clear about your update policy, since some of your tenants may be reluctant to allow changes to their environments, or they might have requirements that limit the times when you can update their service. You need to identify your tenants' requirements, clarify your own requirements to operate your service, find a balance that works for everyone, and then communicate this clearly.

Your customers' requirements

Consider the following questions:

- Do your customers have expectations or requirements about when they can be updated? These might be formally communicated to you in contracts or service-level agreements, or they might be informal.
- Do your customers expect service-defined or even self-defined maintenance windows? They might need to communicate to their own customers about any potential outages.
- Do your customers have any regulatory concerns that require additional approval before updates can be applied? For example, if you provide a health solution that includes IoT components, you might need to get approval from the United States Food and Drug Administration (FDA) before applying an update.
- Are any of your customers particularly sensitive or resistant to having updates applied? Try to understand why. For example, if they run a physical store or a retail website, they may want to avoid updates around Black Friday, as the risks are higher than potential benefits.
- What's your track record of successfully completing updates without any impact to your customers? You should follow good DevOps, testing, deployment, and monitoring practices to reduce the likelihood of outages, and to ensure that you quickly identify any issues that updates introduce. If your customers know that you're able to update their environments smoothly, they're less likely to object.
- Will customers want to roll back updates if there's a breaking change?

Your requirements

You also need to consider the following questions from your own perspective:

- Is it reasonable for your customers to have control over when updates are applied? If you're building a solution used by large enterprise customers, the answer may be yes. However, if you're building a consumer-focused solution, it's unlikely you'll give any control over how you upgrade or operate your solution.
- How many versions of your solution can you reasonably maintain at one time? Remember that if you find a bug and need to hotfix it, you might need to apply the hotfix to all of the versions in use.
- What are the consequences of letting customers fall too far behind the current version? If you release new features on a regular basis, will old versions become obsolete quickly? Also, depending on your upgrade strategy and the types of changes, you might need to maintain separate infrastructures for each version of your solution. So, there might be both operational and financial costs, as you maintain support for older versions.
- Can your deployment strategy support rollbacks to previous versions? Is this something you want to enable?

> **NOTE**
> Consider whether you need to take your solution offline for updates or maintenance. Generally, outage windows are seen as an outdated practice, and modern DevOps practices and cloud technologies enable you to avoid downtime during updates and maintenance. You need to design for this, so it's important to consider your update process when you're designing your solution architecture. Note that even if you don't plan for outages, you might still consider defining a regular maintenance window, so that your customers understand that changes happen during specific times. For more information on achieving zero-downtime deployments, see Achieving no downtime through versioned service updates.

Find a balance

If you leave cadence of your service updates entirely to your tenants' discretion, they may choose to never update. It's important to allow yourself to update your solution, while factoring in any reasonable concerns or constraints that your customers might have. For example, if a customer is particularly sensitive to updates on a Friday (since that's their busiest day of the week), can you just as easily defer updates to Mondays, without impacting your solution?

One approach that can work well is to roll out updates on a tenant-by-tenant basis, using one of the approaches described below. Give your customer notice of planned updates. Allow customers to temporarily opt out, but not forever; put a reasonable limit on when you will require the update to be applied.

Also, consider allowing yourself the ability to deploy security patches, or other critical hotfixes, with minimal or no advance notice.

Another approach can be to allow tenants to initiate their own updates, at a time of their choosing. Again, you should provide a deadline, at which point you apply the update on their behalf.

> **WARNING**
> Be careful about enabling tenants to initiate their own updates. This is complex to implement, and it will require significant development and testing effort to deliver and maintain.

Whatever you do, ensure you have a process to monitor the health of your tenants, especially before and after updates are applied. Often, critical production incidents (also called *live-site incidents*) happen after updates to code or configuration. Therefore, it's important you proactively monitor for and respond to any issues to retain customer confidence. For more information about monitoring, see Monitoring for DevOps

Communicate with your customers

Clear communication is key to building your customers' confidence. It's important to explain the benefits of regular updates, including new features, bug fixes, resolving security vulnerabilities, and performance improvements. One of the benefits of a modern cloud-hosted solution is the ongoing delivery of features and updates.

Consider the following questions:

- Will you notify customers of upcoming updates?
- If you do, will you implicitly request permission by providing an opt-out process?
- Do you have a scheduled maintenance window that you use when you apply updates?
- What if you have an emergency update, like a critical security patch? Can you force updates in those situations?
- If you can't proactively notify customer of upcoming updates, can you provide retrospective notifications? For example, can you update a page on your website with the list of updates that you've applied?

Communicate with your customer support team

It's important that your own support team has full visibility into updates that have been applied to each tenant. Customer support representatives should be able to easily answer the following questions:

- Have updates recently been applied to a tenant's infrastructure?
- What was the nature of those updates?
- What was the previous version?
- How frequently are updates applied to this tenant?

If one of your customers has a problem because of an update, you need to ensure your customer support team has the information necessary to understand what's changed.

Deployment strategies to support updates

Consider how you will deploy updates to your infrastructure. This is heavily influenced by the tenancy model that you use. Three common approaches for deploying updates are deployment stamps, feature flags, and deployment rings.

In all cases, ensure that you have sufficient reporting/visibility, so that you know what version of infrastructure, software, or feature each tenant is on, what they are eligible to migrate to, and any time-related data associated those states.

Deployment stamps

Some multitenant applications are a good fit for the Deployment Stamps pattern, in which you deploy multiple copies of your application and other components. Depending on your isolation requirements, you might deploy a stamp for each tenant, or shared stamps that run multiple tenants' workloads.

Stamps are a great way to provide isolation between tenants. They also provide you with flexibility for your update process, since you can roll out updates progressively across stamps, without affecting others.

Feature flags

Feature flags enable you to add functionality to your solution, while only exposing to a subset of your customers or tenants. You might use feature flags, if you deploy updates regularly but want to avoid showing new functionality, or if you want to avoid applying changes in behavior until a customer opts in.

You can embed feature flag support into your application by writing code yourself, or by using a service like Azure App Configuration.

Deployment rings

Deployment rings enable you to progressively roll out updates across a set of tenants or deployment stamps. You can assign a subset of tenants to each ring. A *canary* ring includes your own test tenants and customers who want to have updates as soon as they are available, with the understanding that they may receive more frequent updates, and that updates might not have been through as comprehensive a validation process as in the other things. An *early adopter* ring contains tenants who are slightly more risk-averse, but who are still prepared to receive regular updates. Most of your tenants will belong to the *users* ring, which receives less frequent and more highly tested updates.

API versions

If your service exposes an external API, consider that any updates you apply might affect the way that customers or partners integrate with your platform. In particular, you need to be conscious of breaking changes to your APIs. Consider using an API versioning strategy to mitigate the risk of updates to your API.

Next steps

- Consider when you would map requests to tenants, in a multitenant solution.
- Review the DevOps checklist in Azure Well-Architected Framework.

Map requests to tenants in a multitenant solution

11/19/2021 • 9 minutes to read • Edit Online

Whenever a request arrives into your application, you need to determine the tenant that the request is intended for. When you have tenant-specific infrastructure that may even be hosted in different geographic regions, you need to match the incoming request to a tenant. Then, you must forward the request to the physical infrastructure that hosts that tenant's resources, as illustrated below:

> **NOTE**
>
> This page mostly discusses HTTP-based applications, like websites and APIs. However, many of same underlying principles apply to multitenant applications that use other communication protocols.

Approaches to identify tenants

There are multiple ways you can identify the tenant for an incoming request.

Domain names

If you use tenant-specific domain or subdomain names, it's likely that requests can be easily mapped to tenants. However, consider the following questions:

- How will users know which domain name to use to access the service?
- Do you have a central entry point, like a landing page or login page, that all the tenants use? If you do, how will users identify the tenant that they need to access?
- What other information are you using to verify access to the tenant, such as authorization tokens? Do the authorization tokens include the tenant-specific domain names?

HTTP request properties

If you don't use tenant-specific domain names, you might still be able to use aspects of the HTTP request to identify the tenant that a particular request is for. For example, consider an HTTP request that identifies the

tenant name as `tailwindtraders`. This might be communicated using the following:

- **The URL path structure**, such as `https://app.contoso.com/tailwindtraders/`.
- **A query string** in the URL, such as `https://contoso.com/app?tenant=tailwindtraders`.
- **A custom HTTP request header**, such as `X-Tenant-Id: tailwindtraders`.

> **IMPORTANT**
>
> Custom HTTP request headers aren't useful where HTTP GET requests are issued from a web browser, or where the requests are handled by some types of web proxy. You should only use custom HTTP headers for GET operations when you're building an API, or if you control the client that issues the request and there's no web proxy included in the request processing chain.

When using this approach, you should consider the following questions:

- Will users know how to access the service? For example, if you use a query string to identify tenants, will a central landing page need to direct users to the correct tenant, by adding the query string?
- Do you have a central entry point, like a landing page or login page, that all tenants use? If you do, how will users identify the tenant that they need to access?
- Does your application provide APIs? For example, is your web application a single-page application (SPA) or a mobile application with an API backend? If it is, you might be able to use an API gateway or reverse proxy to perform tenant mapping.

Token claims

Many applications use claims-based authentication and authorization protocols, such as OAuth 2.0 or SAML. These protocols provide authorization tokens to the client. A token contains a set of *claims*, which are pieces of information about the client application or user. Claims can be used to communicate information like a user's email address. Your system can then include the user's email address, look up the user-to-tenant mapping, and then forward the request to the appropriate physical tenant infrastructure. Or, you might even include the tenant mapping in your identity system, and add a tenant ID claim to the token.

If you are using claims to map requests to tenants, you should consider the following questions:

- Will you use a claim to identify a tenant? Which claim will you use?
- Can a user be a member of multiple tenants? If this is possible, then how will users select the tenants they'd like to work with?
- Is there a central authentication and authorization system for all tenants? If not, how will you ensure that all token authorities issue consistent tokens and claims?

API keys

Many applications expose APIs. These might be for internal use within your organization, or for external use by partners or customers. A common method of authentication for APIs is to use an *API key*. API keys are provided with each request, and they can be used to look up the tenant.

API keys can be generated in several ways. A common approach is to generate a cryptographically random value and store it in a lookup table, alongside the tenant ID. When a request is received, your system finds the API key in the lookup table, and it then matches it to a tenant ID. Another approach is to create a meaningful string with a tenant ID included inside it, and then you would digitally sign the key, by using an approach like HMAC. When you process each request, you verify the signature and then extract the tenant ID.

> **NOTE**
> API keys don't provide a high level of security because they need to be manually created and managed, and because they don't contain claims. A more modern and secure approach is to use a claims-based authorization mechanism with short-lived tokens, such as OAuth 2.0 or OpenID Connect.

Consider the following questions:

- How will you generate and issue API keys?
- How will your API clients securely receive and store the API key?
- Do you need your API keys to expire after a period of time? How will you rotate your clients' API keys, without causing downtime?
- Does just relying on customer-rolled API keys provide an adequate level of security for your APIs?

> **NOTE**
> API keys are not the same as passwords. API keys must be generated by the system, and they must be unique across all the tenants, so that each API key can be uniquely mapped to a single tenant. API gateways, such as Azure API Management, can generate and manage API keys, validate keys on incoming requests, and map keys to individual API subscribers.

Client certificates

Client certificate authentication, sometimes called mutual TLS (mTLS), is commonly used for service-to-service communication. Client certificates provide a secure way to authenticate clients. Similarly to tokens and claims, client certificates provide *attributes* that can be used to determine the tenant. For example, the *subject* of the certificate may contain the email address of the user, which can be used to look up the tenant.

When planning to use client certificates for tenant mapping consider the following:

- How will you safely issue and renew the client certificates that are trusted by your service? Client certificates can be complex to work with, since they require special infrastructure to manage and issue certificates.
- Will client certificates be used only for initial login requests, or attached to all requests to your service?
- Will the process of issuing and managing certificates become unmanageable when you have a large number of clients?
- How will you implement the mapping between the client certificate and the tenant?

Reverse proxies

A reverse proxy, also referred to as an application proxy, can be used to route HTTP requests. A reverse proxy accepts a request from an ingress endpoint, and it can forward the request to one of many backend endpoints. Reverse proxies are useful for multitenant applications since they can perform the mapping between some piece of request information, offloading the task from your application infrastructure.

Many reverse proxies can use the properties of a request to make a decision about tenant routing. They can inspect the destination domain name, URL path, query string, HTTP headers, and even claims within tokens.

The following common reverse proxies are used in Azure:

- Azure Front Door is a global load balancer and web application firewall. It uses the Microsoft global edge network to create fast, secure, and highly scalable web applications.
- Azure Application Gateway is a managed web traffic load balancer that you deploy into the same physical region as your backend service.
- Azure API Management is optimized for API traffic.

- Commercial and open-source technologies (that you host yourself) include nginx, Traefik, and HAProxy.

Request validation

It is important that your application validates that any requests that it receives are authorized for the tenant. For example, if your application uses a custom domain name to map requests to the tenant, then your application must still check that each request received by the application is authorized for that tenant. Even though the request includes a domain name or other tenant identifier, it doesn't mean you should automatically grant access. When you use OAuth 2.0, you perform the validation by inspecting the *audience* and *scope* claims.

> **NOTE**
>
> This is part of the *assume zero trust* security design principle in the Microsoft Azure Well-Architected Framework.

When implementing request validation, you should consider the following:

- How will you authorize all the requests to your application? You need to authorize requests, regardless of the approach you use to map them to physical infrastructure.
- Use trusted and widely used authentication and authorization frameworks and middleware, instead of implementing all of the validation logic yourself. For example, don't build token signature validation logic or client certificate cryptography libraries. Instead, use features of your application platform (or known trusted packages) that have been validated and tested.

Performance

Tenant mapping logic likely runs on every request to your application. Consider how well the tenant mapping process will scale, as your solution grows. For example, if you query a database table as part of your tenant mapping, will the database support a large amount of load? If your tenant mapping requires decrypting a token, will the computation requirements become too high over time? If your traffic is fairly modest, then this isn't likely to affect your overall performance. When you have a high-scale application, though, the overhead involved in this mapping can become significant.

Session affinity

One approach to reducing the performance overhead of tenant mapping logic is to use *session affinity*. Rather than perform the mapping on every request, consider computing the information only on the first request for each session. Your application then provides a *session cookie* to the client that can then passed back to your service, with all subsequent client requests within that session.

> **NOTE**
>
> Many networking and application services in Azure can issue session cookies and natively route requests by using session affinity.

Consider the following questions:

- Can you use session affinity to reduce the overhead of mapping requests to tenants?
- What services do you use to route requests to the physical deployments for each tenant? Do these services support cookie-based session affinity?

Tenant migration

Tenants often need to be moved to new infrastructure as part of the tenant lifecycle. When a tenant is moved to a new deployment, the HTTP endpoints they access might change. When this happens, consider that your

tenant-mapping process needs to be updated. You may need to consider the following:

- If your application uses domain names for mapping requests, then it might also require a DNS change at the time of the migration. The DNS change might take time to propagate to clients, depending on the time-to-live of the DNS entries in your DNS service.
- If your migration changes the addresses of any endpoints during the migration process, then consider temporarily redirecting requests for the tenant to a maintenance page that automatically refreshes.

Next steps

Learn about considerations when you work with domain names in a multitenant application.

Considerations when using domain names in a multitenant solution

11/19/2021 • 9 minutes to read • Edit Online

In many multitenant web applications, a domain name can be used as a way to identify a tenant, to help with routing requests, and to provide a branded experience to your customers. Two common approaches are to use subdomains and custom domain names.

Subdomains

Each tenant might get a unique subdomain under a common shared domain name. Let's consider an example multitenant solution built by Contoso. Customers purchase Contoso's product to help manage their invoice generation. All of Contoso's tenants might be assigned their own subdomain, under the `contoso.com` domain name. Or, if you use regional deployments, you might assign subdomains under the `us.contoso.com` and `eu.contoso.com` domains. In this article, we refer to these as *stem domains*. Each customer gets their own subdomain under your stem domain. For example, Tailwind Toys might be assigned `tailwind.contoso.com`, and Adventure Works might be assigned `adventureworks.contoso.com`.

> **NOTE**
>
> Many Azure services use this approach. For example, when you create an Azure storage account, it is assigned a set of subdomains for you to use, such as `<your account name>.blob.core.windows.net`.

Manage your domain namespace

When you create subdomains under your own domain name, you need to be mindful that you could have multiple customers with similar names. Since they share a single stem domain, the first customer to get a particular domain will get their preferred name, and subsequent customers will have to use alternate subdomain names, since full domain names must be globally unique.

Wildcard DNS

Consider using wildcard DNS entries to simplify the management of subdomains. Instead of creating DNS entries for `tailwind.contoso.com`, `adventureworks.contoso.com`, and so forth, you could instead create a wildcard entry for `*.contoso.com` and direct all subdomains to single IP address (A record) or canonical name (CNAME record).

> **NOTE**
>
> Make sure that your web-tier services support wildcard DNS, if you plan to rely on this feature. Many Azure services, including Azure Front Door and Azure App Service, support wildcard DNS entries.

Subdomains with multipart stem domains

Many multitenant solutions are spread across multiple physical deployments. This is common, when you need to comply with data residency requirements, or when you want to provide better performance by deploying resources geographically closer to the users. Even within a single region, you might also need to spread your tenants across independent deployments, as part of your scaling strategy. If you plan to use subdomains for each tenant, you might consider a multipart subdomain structure.

Here's an example: Contoso publishes a multitenant application for its four customers. Adventure Works and

Tailwind Traders are in the United States, and their data is stored on a shared US instance of the Contoso platform. Fabrikam and Worldwide Importers are in Europe, and their data is stored on a European instance.

If Contoso chose to use a single stem domain, **contoso.com**, for all their customers, here's what this might look like:

The DNS entries (that are required to support this configuration) might look like this:

SUBDOMAIN	CNAME TO
`adventureworks.contoso.com`	`us.contoso.com`
`tailwind.contoso.com`	`us.contoso.com`
`fabrikam.contoso.com`	`eu.contoso.com`
`worldwideimporters.contoso.com`	`eu.contoso.com`

Each new customer that is onboarded requires a new subdomain, and the number of subdomains grows with each customer.

Alternatively, Contoso could use deployment- or region-specific stem domains, like this:

The DNS entries for this deployment might look like this:

SUBDOMAIN	CNAME TO
`*.us.contoso.com`	`us.contoso.com`
`*.eu.contoso.com`	`eu.contoso.com`

Contoso doesn't need to create subdomain records for every customer. Instead, they have a single wildcard DNS record for each geography's deployment, and any new customers who are added underneath that stem will automatically inherit the CNAME record.

There are benefits and drawbacks to each approach. When using a single stem domain, each tenant you onboard requires a new DNS record to be created, which introduces more operational overhead. However, you have more flexibility, if you need to move tenants between deployments, since you can change the CNAME record to direct their traffic to another deployment. This won't affect any other tenants. When using multiple

stem domains, there's a lower management overhead, and you can reuse customer names across multiple regional stem domains, since each one effectively represents its own namespace.

Custom domain names

You might want to enable your customers to bring their own domain names. Some customers see this as an important aspect of their branding. It might also be required to meet customers' security requirements, especially if they need to supply their own TLS certificates. While it might seem trivial to enable customers to bring their own domain names, there are some hidden complexities to this approach, and it requires thoughtful consideration.

Name resolution

Ultimately, each domain name needs to be resolved to an IP address. As you've seen, the approach by which this happens can depend on whether you deploy a single instance or multiple instances of your solution.

Let's return to our example. One of Contoso's customers, Fabrikam, has asked to use `invoices.fabrikam.com`, as the custom domain name to access Contoso's service. Since Contoso has multiple deployments of their platform, they decide to use subdomains and CNAME records to achieve their routing requirements. Contoso and Fabrikam configure the following DNS records:

NAME	RECORD TYPE	VALUE	CONFIGURED BY
`invoices.fabrikam.com`	CNAME	`fabrikam.eu.contoso.com`	Fabrikam
`*.eu.contoso.com`	CNAME	`eu.contoso.com`	Contoso
`eu.contoso.com`	A	(Contoso's IP address)	Contoso

From a name resolution perspective, this chain of records accurately resolves requests for `invoices.fabrikam.com`, to the IP address of Contoso's European deployment.

Host header resolution

Name resolution is only half of the problem. All of the web components (within Contoso's European deployment) need to be aware of how to handle requests that arrive with Fabrikam's domain name in their `Host` request header. Depending on the specific web technologies that Contoso uses, this may require further configuration for each tenant's domain name, which adds extra operational overhead to the onboarding of tenants.

You can also consider rewriting host headers, so that regardless of the incoming request's `Host` header, your web server sees a consistent header value. For example, Azure Front Door enables you to rewrite `Host` headers, so that regardless of the request, your application server receives a single `Host` header. Azure Front Door propagates the original host header in the `X-Forwarded-Host` header, so that your application can inspect it, to resolve the tenant.

Domain validation

It's important to validate the ownership of custom domains before onboarding them. Otherwise, you risk a customer accidentally or maliciously *parking* a domain name.

Let's consider Contoso's onboarding process for Adventure Works, who have asked to use `invoices.adventureworks.com` as their custom domain name. Unfortunately, somebody made a typo when they tried to onboard the custom domain name, and they missed the *s*. So, they set it up as `invoices.adventurework.com`. Not only does the traffic not flow correctly, but when another company named *Adventure Work* tries to add their custom domain to Contoso's platform, they're told the domain name is already in use.

When working with custom domains, especially within a self-service or automated process, it's common to require a domain verification step. This might require that the CNAME records be set up before the domain can be added. Alternatively, Contoso might generate a random string and ask Adventure Works to add a DNS TXT record with the string value. That would prevent the domain name from being added, until the verification is completed.

Dangling DNS and subdomain takeover attacks

When you work with custom domain names, you are potentially vulnerable to a class of attack called *dangling DNS* or *subdomain takeover*. This attack happens when customers disassociate their custom domain name from your service, but they don't delete the record from their DNS server. This DNS entry then points to a non-existent resource and is vulnerable to a takeover.

Let's consider how Fabrikam's relationship with Contoso might change:

1. Fabrikam has decided to no longer work with Contoso, and so they have terminated their business relationship.
2. Contoso has offboarded the Fabrikam tenant, and they requested for `fabrikam.contoso.com` to no longer work. However, Fabrikam forgot to delete the CNAME record for `invoices.fabrikam.com`.
3. A malicious actor creates a new Contoso account and gives it the name `fabrikam`.
4. The attacker onboards the custom domain name `invoices.fabrikam.com` to their new tenant. Since Contoso performs CNAME-based domain validation, they check Fabrikam's DNS server. They see that the DNS server returns a CNAME record for `invoices.fabrikam.com`, which points to `fabrikam.contoso.com`. Contoso considers the custom domain validation to be successful.
5. If any Fabrikam employees tried to access the site, requests would appear to work. If the attacker sets up their Contoso tenant with Fabrikam's branding, employees might be fooled into accessing the site and providing sensitive data, which the attacker can then access.

A common strategy to protect against dangling DNS attacks is to require that the CNAME record is deleted, before the domain name can be removed from the tenant's account. You could also consider prohibiting the reuse of tenant identifiers, and you can then strengthen your custom domain onboarding process by using a randomly generated TXT record (that is different for each onboarding attempt).

TLS/SSL certificates

Transport Layer Security (TLS) is an essential component, when working with modern applications. It provides trust and security to your web applications. The ownership and management of TLS certificates is something that needs careful consideration, for multitenant applications.

Typically, the owner of a domain name will be responsible for issuing and renewing its certificates. For example, Contoso is responsible for issuing and renewing TLS certificates for `us.contoso.com`, as well as a wildcard certificate for `*.contoso.com`. Similarly, Fabrikam would generally be responsible for managing any records for the `fabrikam.com` domain, including `invoices.fabrikam.com`. The CAA (Certificate Authority Authorization) DNS record type can be used by a domain owner, to ensure that only specific authorities can create certificates for their domain.

If you plan to allow customers to bring their own domains, consider whether you plan to issue the certificate on the customer's behalf, or whether the customers must bring their own certificates. Each option has benefits and drawbacks. If you issue a certificate for a customer, you can handle the renewal of the certificate, so the customer doesn't have to remember to keep it updated. However, if the customers have CAA records on their domain names, they might need to authorize you to issue certificates on their behalf. If you expect customers should issue and provide you with their own certificates, you are responsible for receiving and managing the private keys in a secure manner, and you might have to remind your customers to renew the certificate before it expires, to avoid an interruption in their service.

Several Azure services support automatic management of certificates for custom domains. For example, Azure Front Door and App Service provide certificates for custom domains, and they automatically handle the renewal process. This removes the burden of managing certificates, from your operations team. However, you still need to consider the question of ownership and authority, such as whether CAA records are in effect and configured correctly. Also, you need to ensure your customers' domains are configured to allow the certificates that are managed by the platform.

Next steps

Return to the architectural considerations overview. Or, review the Microsoft Azure Well-Architected Framework.

Architectural approaches for storage and data in multitenant solutions

11/19/2021 • 15 minutes to read • Edit Online

When planning multitenant storage or data components, you need to decide on an approach for sharing or isolating your tenants' data. Data is often considered the most valuable part of a solution, since it represents your or your customers' valuable business information. So, it's important to carefully plan the approach you use to manage data in a multitenant environment. On this page, we provide guidance about the key considerations and requirements that are essential for solution architects when deciding on an approach to store data in a multitenant system. We then suggest some common patterns for applying multitenancy to storage and data services, and some antipatterns to avoid. Finally, we provide targeted guidance for some specific situations.

Key considerations and requirements

It's important to consider the approaches you use for storage and data services from a number of perspectives, which approximately align to the pillars of the Azure Well-Architected Framework.

Scale

When working with services that store your data, you should consider the number of tenants you have, and the volume of data you store. If you have a small number of tenants (such as five or less), and you're storing small amounts of data for each tenant, then it's likely to be a wasted effort to plan a highly scalable data storage approach, or to build a fully automated approach to manage your data resources. As you grow, you will increasingly benefit from having a clear strategy to scale your data and storage resources, and to apply automation to their management. When you have 50 tenants or more, or if you plan to reach that level of scale, then it's especially important to design your data and storage approach, with scale as a key consideration.

Consider the extent to which you plan to scale, and clearly plan your data storage architectural approach to meet that level of scale.

Performance predictability

Multitenant data and storage services are particularly susceptible to the Noisy Neighbor problem. It's important to consider whether your tenants could affect each other's performance. For example, do your tenants have overlapping peaks in their usage patterns over time? Do all of your customers use your solution at the same time each day, or are requests distributed evenly? Those factors will impact the level of isolation you need to design for, the amount of resources you need to provision, and the degree to which resources can be shared between tenants.

It's important to consider Azure's resource and request quotas as part of this decision. For example, suppose you deploy a single storage account to contain all of your tenants' data. If you exceed a specific number of storage operations per second, Azure Storage will reject your application's requests, and all of your tenants will be impacted. This is called *throttling* behavior. It's important that you monitor for throttled requests. See Retry guidance for Azure services for further information.

Data isolation

When designing a solution that contains multitenant data services, there are usually different options and levels of data isolation, each with their own benefits and tradeoffs. For example:

- When using Azure Cosmos DB, you can deploy separate containers for each tenant, and you can share databases and accounts between multiple tenants. Alternatively, you might consider deploying different databases or even accounts for each tenant, depending on the level of isolation required.

- When using Azure Storage for blob data, you can deploy separate blob containers for each tenant, or you can deploy separate storage accounts.
- When using Azure SQL, you can use separate tables in shared databases, or you can deploy separate databases or servers for each tenant.
- In all Azure services, you can consider deploying resources within a single shared Azure subscription, or you can use multiple Azure subscriptions--perhaps even one per tenant.

There is no single solution that works for every situation. The option you choose depends on a number of factors and the requirements of your tenants. For example, if your tenants need to meet specific compliance or regulatory standards, you might need to apply a higher level of isolation. Similarly, you might have commercial requirements to physically isolate your customers' data, or you might need to enforce isolation to avoid the Noisy Neighbor problem. Additionally, if tenants need to use their own encryption keys, they have individual backup and restore policies, or they need to have their data stored in different geographical locations, you might need to isolate them from other tenants, or group them with tenants that have similar policies.

Complexity of implementation

It's important to consider the complexity of your implementation. It's good practice to keep your architecture as simple as possible, while still meeting your requirements. Avoid committing to an architecture that will become increasingly complex as you scale, or an architecture that you don't have the resources or expertise to develop and maintain.

Similarly, if your solution doesn't need to scale to a large number of tenants, or if you don't have concerns around performance or data isolation, then it's better to keep your solution simple and avoid adding unnecessary complexity.

A particular concern for multitenant data solutions is the level of customization you support. For example, can a tenant extend your data model or apply custom data rules? Ensure you design for this upfront. Avoid forking or providing custom infrastructure for individual tenants, since this inhibits your ability to scale, to test your solution, and to deploy updates. Instead, consider using feature flags and other forms of tenant configuration.

Complexity of management and operations

Consider how you plan to operate your solution, and how your multitenancy approach affects your operations and processes. For example:

- Consider cross-tenant management operations, such as regular maintenance activities. If you use multiple accounts, servers, or databases, how will you initiate and monitor the operations for each tenant?
- If you monitor or meter your tenants, consider how your solution reports metrics, and whether they can be easily linked to the tenant that triggered the request.
- Reporting data from across isolated tenants may require that each tenant publishes data to a centralized data warehouse, rather than running queries on each database individually and then aggregating the results.
- If you use a database that enforces a schema, plan how you will deploy schema updates across your estate. Consider how your application knows which schema version to use for a specific tenant's database queries.
- Consider your tenants' high availability requirements (for example, uptime service level agreements, or SLAs) and disaster recovery requirements (for example, recovery time objectives, or RTOs, and recovery point objectives, or RPOs). If tenants have different expectations, will you be able to meet each tenant's requirements?
- How will you migrate tenants if they need to move to a different type of service, a different deployment, or another region?

Cost

Generally, the higher the density of tenants to your deployment infrastructure, the lower the cost to provision that infrastructure. However, shared infrastructure increases the likelihood of issues like the Noisy Neighbor problem, so consider the tradeoffs carefully.

Approaches and patterns to consider

Several design patterns from the Azure Architecture Center are of relevance to mulitenant storage and data services. You might choose to follow one pattern consistently. Or, you could consider mixing and matching patterns. For example, you might use a multitenant database for most of your tenants, but deploy single-tenant stamps for tenants who pay more or who have unusual requirements. Similarly, it's often a good practice to scale by using deployment stamps, even when you use a multitenant database or sharded databases within a stamp.

Deployment Stamps pattern

The Deployment Stamps pattern involves deploying dedicated infrastructure for a tenant or group of tenants. A single stamp might contain multiple tenants or might be dedicated to a single tenant.

When using single-tenant stamps, the Deployment Stamps pattern tends to be straightforward to implement, because each stamp is likely to be unaware of any other, so no multitenancy logic or capabilities need to be built into the application layer. When each tenant has their own dedicated stamp, this pattern provides the highest degree of isolation, and it mitigates the Noisy Neighbor problem. It also provides the option for tenants to be configured or customized according to their own requirements, such as to be located in a specific geopolitical region or to have specific high availability requirements.

When using multitenant stamps, other patterns need to be considered to manage multitenancy within the stamp, and the Noisy Neighbor problem still might apply. However, by using the Deployment Stamps pattern, you can ensure that you can continue to scale as your solution grows.

The biggest problem with the Deployment Stamps pattern, when being used to serve a single tenant, tends to be the cost of the infrastructure. When using single-tenant stamps, each stamp needs to have its own separate set of infrastructure, which isn't shared with other tenants. You also need to ensure that the resources deployed for a stamp are sufficient to meet the peak load for that tenant's workload. Ensure that your pricing model offsets the cost of deployment for the tenant's infrastructure.

Single-tenant stamps often work well when you have a small number of tenants. As your number of tenants grows, it's possible but increasingly difficult to manage a fleet of stamps (see this case study as an example). You can also apply the Deployment Stamps pattern to create a fleet of multitenant stamps, which can provide benefits for resource and cost sharing.

To implement the Deployment Stamps pattern, it's important to use automated deployment approaches. Depending on your deployment strategy, you might consider managing your stamps within your deployment pipelines, by using declarative infrastructure as code, such as Bicep, ARM templates, or Terraform templates. Alternatively, you might consider building custom code to deploy and manage each stamp, such as by using the Azure SDKs.

Shared multitenant databases and file stores

You might consider deploying a shared multitenant database, storage account, or file share, and sharing it across all of your tenants.

This approach provides the highest density of tenants to infrastructure, so it tends to come at the lowest cost of any approach. It also often reduces the management overhead, since there's a single database or resource to manage, back up, and secure.

However, when you work with shared infrastructure, there are several caveats to consider:

- When you rely on a single resource, consider the supported scale and limits of that resource. For example, the maximum size of one database or file store, or the maximum throughput limits, will eventually become a hard blocker, if your architecture relies on a single database. Carefully consider the maximum scale you need to achieve, and compare it to your current and future limits, before you select this pattern.
- The Noisy Neighbor problem might become a factor, especially if you have tenants that are particularly busy or generate higher workloads than others. Considering applying the Throttling pattern or the Rate Limiting pattern to mitigate these effects.
- You might have difficulty monitoring the activity and measuring the consumption for a single tenant. Some services, such as Azure Cosmos DB, provide reporting on resource usage for each request, so this information can be tracked to measure the consumption for each tenant. Other services don't provide the same level of detail. For example, the Azure Files metrics for file capacity are available per file share dimension, only when you use premium shares. However, the standard tier provides the metrics only at the storage account level.
- Tenants may have different requirements for security, backup, availability, or storage location. If these don't match your single resource's configuration, you might not be able to accommodate them.
- When working with a relational database, or another situation where the schema of the data is important, then tenant-level schema customization is difficult.

Sharding pattern

The Sharding pattern involves deploying multiple separate databases, called *shards*, that contain one or more tenants' data. Unlike deployment stamps, shards don't imply that the entire infrastructure is duplicated. You might shard databases without also duplicating or sharding other infrastructure in your solution.

Sharding is closely related to *partitioning*, and the terms are often used interchangeably. Consider the Horizontal, vertical, and functional data partitioning guidance.

The Sharding pattern can scale to very large numbers of tenants. Additionally, depending on your workload, you might be able to achieve a high density of tenants to shards, so the cost can be attractive. The Sharding pattern can also be used to address Azure subscription and service quotas, limits and constraints.

Some data stores, such as Azure Cosmos DB, provide native support for sharding or partitioning. When working with other solutions, such as Azure SQL, it can be more complex to build a sharding infrastructure and to route requests to the correct shard, for a given tenant.

Multitenant app with dedicated databases for each tenant

Another common approach is to deploy a single multitenant application, with dedicated databases for each tenant.

In this model, each tenant's data is isolated from the others, and you might be able to support some degree of customization for each tenant.

Because you provision dedicated data resources for each tenant, the cost for this approach can be higher than shared hosting models. However, Azure provides several options you can consider, in order to share the cost of hosting individual data resources across multiple tenants. For example, when you work with Azure SQL, you can consider elastic pools. For Azure Cosmos DB, you can provision throughput for a database and the throughput is shared between the containers in that database, although this approach is not appropriate when you need guaranteed performance for each container.

In this approach, because only the data components are deployed individually for each tenant, you likely can achieve high density for the other components in your solution and reduce the cost of those components.

It's important to use automated deployment approaches when you provision databases for each tenant.

Geodes pattern

The Geode pattern is designed specifically for geographically distributed solutions, including multitenant solutions. It supports high load and high levels of resiliency. When working with the Geode pattern, the data tier must be able to replicate the data across geographic regions, and it should support multi-geography writes.

Azure Cosmos DB provides multi-master writes to support this pattern, and Cassandra supports multi-region clusters. Other data services are generally not able to support this pattern, without significant customization.

Antipatterns to avoid

When working with multitenant data services, it's important to avoid situations that inhibit your ability to scale.

For relational databases, these include:

- **Table-based isolation.** When you work within a single database, avoid creating individual tables for each tenant. A single database won't be able to support very large numbers of tenants when you use this approach, and it becomes increasingly difficult to query, manage, and update data. Instead, consider using a single set of multitenant tables with a tenant identifier column. Alternatively, you can use one of the patterns described above to deploy separate databases for each tenant.
- **Column-level tenant customization.** Avoid applying schema updates that only apply to a single tenant. For example, suppose you have a single multitenant database. Avoid adding a new column to meet a specific tenant's requirements. It might be acceptable for a small number of customizations, but this rapidly becomes unmanageable when you have a large number of customizations to consider. Instead, consider revising your data model to track custom data for each tenant in a dedicated table.
- **Manual schema changes.** Avoid updating your database schema manually, even if you only have a single shared database. It's easy to lose track of the updates you've applied, and if you need to scale out to more databases, it's challenging to identify the correct schema to apply. Instead, build an automated pipeline to deploy your schema changes, and use it consistently. Track the schema version used for each tenant in a dedicated database or lookup table.
- **Version dependencies.** Avoid having your application take a dependency on a single version of your database schema. As you scale, you may need to apply schema updates at different times for different tenants. Instead, ensure your application version is backwards-compatible with at least one schema version, and avoid destructive schema updates.

Databases

There are some features that can be useful for multitenancy. However, these aren't available in all database services. Consider whether you need these, when you decide on the service to use for your scenario:

- **Row-level security** can provide security isolation for specific tenants' data in a shared multitenant database. This feature is available in Azure SQL and Postgres Flex, but it's not available in other databases, like MySQL or Azure Cosmos DB.
- **Tenant-level encryption** might be required to support tenants that provide their own encryption keys for their data. This feature is available in Azure SQL as part of Always Encrypted. Cosmos DB provides customer-managed keys at the account level and also supports Always Encrypted.
- **Resource pooling** provides the ability to share resources and cost, between multiple databases or containers. This feature is available in Azure SQL's elastic pools and managed instances and in Azure Cosmos DB's database throughput, although each option has limitations you should be aware of.
- **Sharding and partitioning** has stronger native support in some services than others. This feature is available in Azure Cosmos DB, by using its logical and physical partitioning, and in Postgres Hyperscale. While Azure SQL doesn't natively support sharding, it provides sharding tools to support this type of architecture.

Additionally, when working with relational databases or other schema-based databases, consider where the schema upgrade process should be triggered, when you maintain a fleet of databases. In a small estate of databases, you might consider using a deployment pipeline to deploy schema changes. As you grow, it might be better for your application tier to detect the schema version for a specific database and to initiate the upgrade process.

File and blob storage

Consider the approach you use to isolate data within a storage account. For example, you might deploy separate storage accounts for each tenant, or you might share storage accounts and deploy individual containers. Alternatively, you might create shared blob containers, and then you can use the blob path to separate data for each tenant. Consider Azure subscription limits and quotas, and carefully plan your growth to ensure your Azure

resources scale to support your future needs.

If you use shared containers, plan your authentication and authorization strategy carefully, to ensure that tenants can't access each other's data. Consider the Valet Key pattern, when you provide clients with access to Azure Storage resources.

Cost allocation

Consider how you'll measure consumption and allocate costs to tenants, for the use of shared data services. Whenever possible, aim to use built-in metrics instead of calculating your own. However, with shared infrastructure, it becomes hard to split telemetry for individual tenants. Application-level custom metering needs to be considered.

In general, cloud-native services, like Azure Cosmos DB and Azure Blob Storage, provide more granular metrics to track and model the usage for a specific tenant. For example, Azure Cosmos DB provides the consumed throughput for every request and response.

Next steps

For more information about multitenancy and specific Azure services, see:

- Multitenancy and Azure Storage
- Multitenancy and Azure SQL Database
- Multitenancy and Azure Cosmos DB

Multitenancy and Azure Storage

11/19/2021 • 11 minutes to read • Edit Online

Azure Storage is a foundational service used in almost every solution. Multitenant solutions often use Azure Storage for blob, file, queue, and table storage. On this page, we describe some of the features of Azure Storage that are useful for multitenant solutions, and then we provide links to the guidance that can help you, when you're planning how you're going to use Azure Storage.

Features of Azure Storage that support multitenancy

Azure Storage includes many features that support multitenancy.

Shared access signatures

When you work with Azure Storage from a client application, it's important to consider whether client requests should be sent through another component that you control, like a content delivery network or API, or if the client should connect directly to your storage account. There might be good reasons to send requests through another component, including caching data at the edge of your network. However, in some situations, it's advantageous for client endpoints to connect directly to Azure Storage to download or upload data. This connection helps you improve the performance of your solution, especially when you work with large blobs or large numbers of files. It also reduces the load on your backend applications and servers, and it reduces the number of network hops. A shared access signature (SAS) enables you to securely provide your client applications with access to objects in Azure Storage.

Shared access signatures can be used to restrict the scope of operations that a client can perform, and the objects that they can perform operations against. For example, if you have a shared storage account for all of your tenants, and you store all of tenant A's data in a blob container named `tenanta`, you can create an SAS that only permits tenant A's users to access that container. For more information, see Isolation models to explore the approaches you can use to isolate your tenants' data in a storage account.

The Valet Key pattern is useful as a way to issue constrained and scoped shared access signatures from your application tier. For example, suppose you have a multitenant application that allows users to upload videos. Your API or application tier can authenticate the client using your own authentication system. You can then provide a SAS to the client that allows them to upload a video file to a specified blob, into a container and blob path that you specify. The client then uploads the file directly to the storage account, avoiding the extra bandwidth and load on your API. If they try to read data from the blob container, or if they try to write data to a different part of the container to another container in the storage account, Azure Storage blocks the request. The signature expires after a configurable time period.

Stored access policies extend the SAS functionality, which enables you to define a single policy that can be used when issuing multiple shared access signatures.

Identity-based access control

Azure Storage also provides identity-based access control by using Azure Active Directory (Azure AD). This capability also enables you to use attribute-based access control, which gives you finer-grained access to blob paths, or to blobs that have been tagged with a specific tenant ID.

Lifecycle management

When you use blob storage in a multitenant solution, your tenants might require different policies for data retention. When you store large volumes of data, you might also want to configure the data for a specific tenant to automatically be moved to the cool or archive storage tiers, for cost-optimization purposes.

Consider using lifecycle management policies to set the blob lifecycle for all tenants, or for a subset of tenants. A lifecycle management policy can be applied to blob containers, or to a subset of blobs within a container. However, there are limits on the number of rules you can specify in a lifecycle management policy. Make sure you plan and test your use of this feature in a multitenant environment, and consider deploying multiple storage accounts, if you will exceed the limits.

Immutable storage

When you configure immutable blob storage on storage containers with time-based retention policies, Azure Storage prevents deletion or modification of the data before a specified time. The prevention is enforced at the storage account layer and applies to all users. Even your organization's administrators can't delete immutable data.

Immutable storage can be useful when you work with tenants that have legal or compliance requirements to maintain data or records. However, you should consider how this feature is used within the context of your tenant lifecycle. For example, if tenants are offboarded and request the deletion of their data, you might not be able to fulfil their requests. If you use immutable storage for your tenants' data, consider how you address this issue in your terms of service.

Server-side copy

In a multitenant system, there is sometimes a need to move data from one storage account to another. For example, if you move a tenant between deployment stamps or rebalance a sharded set of storage accounts, you need to copy or move a specific tenant's data. When working with large volumes of data, it's advisable to use server-side copy APIs to decrease the time it takes to migrate the data.

The AzCopy tool is an application that you can run from your own computer, or from a virtual machine, to manage the copy process. AzCopy is compatible with the server-side copy feature, and it provides a scriptable command-line interface that you can run from your own solutions. AzCopy is also helpful for uploading and downloading large volumes of data.

If you need to use the server-side copy APIs directly from your code, consider using the Put Block From URL API, Put Page From URL API, Append Block From URL API, and the Copy Blob From URL API when working with smaller blobs.

Object replication

The Object replication feature automatically replicates data between a source and destination storage account. Object replication is asynchronous. In a multitenant solution, this feature can be useful when you need to continuously replicate data between deployment stamps, or in an implementation of the Geode pattern.

Encryption

Azure Storage enables you to provide encryption keys for your data. In a multitenant solution, consider combining this capability with encryption scopes, which enable you to define different encryption keys for different tenants, even if their data is stored in the same storage account. By using these features together, you can also provide tenants with control over their own data. If they need to deactivate their account, they can delete the encryption key and their data is no longer accessible.

Monitoring

When working with a multitenant solution, consider whether you need to measure the consumption for each tenant, and define the specific metrics you need to track, such as the amount of storage used for each tenant (the capacity), or the number of operations performed for each tenant's data.

Azure Storage provides built-in monitoring capabilities. It's important to consider the services you'll use within the Azure Storage account. For example, when you work with blobs, it's possible to view the total capacity of a storage account, but not a single container. In contrast, when you work with file shares, it's possible to see the capacity for each share, but not for each folder.

You can also log all of the requests made to Azure Storage, and then you can aggregate and analyze those logs.

This provides more flexibility in how you aggregate and group data for each tenant. However, in solutions that create high volumes of requests to Azure Storage, it's important to consider whether the benefit you gain from this approach justifies the cost involved in capturing and processing those logs.

Azure Storage inventory provides another approach to measure the total size of a blob container.

Isolation models

When working with a multitenant system using Azure Storage, you need to make a decision about the level of isolation you want to use. Azure Storage supports several isolation models.

Storage accounts per tenant

The strongest level of isolation is to deploy a dedicated storage account for a tenant. This ensures that all storage keys are isolated and can be rotated independently. This approach enables you to scale your solution to avoid limits and quotas that are applicable to each storage account, but you also need to consider the maximum number of storage accounts that can be deployed into a single Azure subscription.

> **NOTE**
>
> Azure Storage has many quotas and limits that you should consider when you select an isolation model. These include Azure service limits, scalability targets, and scalability targets for the Azure Storage resource provider.

Additionally, each component of Azure Storage provides further options for tenant isolation.

Blob storage isolation models

Shared blob containers

When working with blob storage, you might choose to use a shared blob container, and you might then use blob paths to separate data for each tenant:

TENANT ID	EXAMPLE BLOB PATH
`tenant-a`	`https://contoso.blob.core.windows.net/sharedcontainer/tenant-a/blob1.mp4`
`tenant-b`	`https://contoso.blob.core.windows.net/sharedcontainer/tenant-b/blob2.mp4`

While this approach is simple to implement, in many scenarios, blob paths don't provide sufficient isolation across tenants. This is because blob storage doesn't typically provide a concept of directories or folders. This means you can't assign access to all blobs within a specified path. However, Azure Storage provides a capability to list (enumerate) blobs that begin with a specified prefix, which can be helpful when you work with shared blob containers and don't require directory-level access control.

Azure Storage's hierarchical namespace feature provides the ability to have a stronger concept of a directory or folder, including directory-specific access control. This can be useful in some multitenant scenarios where you have shared blob containers, but you want to grant access to a single tenant's data.

In some multitenant solutions, you might only need to store a single blob or set of blobs for each tenant, such as tenant icons for customizing a user interface. In these scenarios, a single shared blob container might be sufficient. You could use the tenant identifier as the blob name, and then read a specific blob instead of enumerating a blob path.

When you work with shared containers, consider whether you need to track the data and Azure Storage service usage for each tenant, and plan an approach to do so. See Monitoring for further information.

Blob containers per tenant

You can create individual blob containers for each tenant within a single storage account. There is no limit to the number of blob containers that you can create, within a storage account.

By creating containers for each tenant, you can use Azure Storage access control, including SAS, to manage access for each tenant's data. You can also easily monitor the capacity that each container uses.

File storage isolation models

Shared file shares

When working with file shares, you might choose to use a shared file share, and then you might use file paths to separate data for each tenant:

TENANT ID	EXAMPLE FILE PATH
`tenant-a`	`https://contoso.file.core.windows.net/share/tenant-a/blob1.mp4`
`tenant-b`	`https://contoso.file.core.windows.net/share/tenant-b/blob2.mp4`

When you use an application that can communicate using the Server Message Block (SMB) protocol, and when you use Active Directory Domain Services either on-premises or in Azure, file shares support authorization at both the share and the directory/file levels.

In other scenarios, consider using SAS to grant access to specific file shares or files. When you use SAS, you can't grant access to directories.

When you work with shared file shares, consider whether you need to track the data and Azure Storage service usage for each tenant, and then plan an approach to do so (as necessary). See Monitoring for further information.

File shares per tenant

You can create individual file shares for each tenant, within a single storage account. There is no limit to the number of file shares that you can create within a storage account.

By creating file shares for each tenant, you can use Azure Storage access control, including SAS, to manage access for each tenant's data. You can also easily monitor the capacity each file share uses.

Table storage isolation models

Shared tables with partition keys per tenant

When using table storage with a single shared table, you can consider using the built-in support for partitioning. Each entity must include a partition key. A tenant identifier is often a good choice for a partition key.

Shared access signatures and policies enable you to specify a partition key range, and Azure Storage ensures that requests containing the signature can only access the specified partition key ranges. This enables you to implement the Valet Key pattern, which allows untrusted clients to access a single tenant's partition, without affecting other tenants.

For high-scale applications, consider the maximum throughput of each table partition and the storage account.

Tables per tenant

You can create individual tables for each tenant within a single storage account. There is no limit to the number of tables that you can create within a storage account.

By creating tables for each tenant, you can use Azure Storage access control, including SAS, to manage access for each tenant's data.

Queue storage isolation models

Shared queues

If you choose to share a queue, consider the quotas and limits that apply. In solutions with a high request volume, consider whether the target throughput of 2,000 messages per second is sufficient.

Queues don't provide partitioning or subqueues, so data for all tenants could be intermingled.

Queues per tenant

You can create individual queues for each tenant within a single storage account. There is no limit to the number of queues that you can create within a storage account.

By creating queues for each tenant, you can use Azure Storage access control, including SAS, to manage access for each tenant's data.

When you dynamically create queues for each tenant, consider how your application tier will consume the messages from each tenant's queue. For more advanced scenarios, consider using Azure Service Bus, which supports features such as topics and subscriptions, sessions, and message auto-forwarding, which can be useful in a multitenant solution.

Next steps

Review Resources for architects and developers of multitenant solutions.

Multitenancy and Azure SQL Database

11/19/2021 • 2 minutes to read • Edit Online

Multitenant solutions on Azure commonly use Azure SQL Database. On this page, we describe some of the features of Azure SQL Database that are useful when working with multitenanted systems, and we link to guidance and examples for how to use Azure SQL in a multitenant solution.

Guidance

The Azure SQL Database team has published extensive guidance on implementing multitenant architectures with Azure SQL Database. See Multi-tenant SaaS patterns with Azure SQL Database. Also, consider the guidance for partitioning Azure SQL databases.

Features of Azure SQL Database that support multitenancy

Azure SQL Database includes a number of features that support multitenancy.

Elastic pools

Elastic pools enable you to share compute resources between a number of databases on the same server. By using elastic pools, you can achieve performance elasticity for each database, while also achieving cost efficiency by sharing your provisioned resources across databases. Elastic pools provide built-in protections against the Noisy Neighbor problem.

More information:

- SQL Database elastic pools
- Resource management in dense elastic pools
- Disaster recovery strategies for applications using SQL Database elastic pools

Elastic database tools

The Sharding pattern enables you to scale your workload across multiple databases. Azure SQL Database provides tools to support sharding. These tools include the management of *shard maps* (a database that tracks the tenants assigned to each shard), as well as initiating and tracking queries and management operations on multiple shards by using *elastic jobs*.

More information:

- Multi-tenant applications with elastic database tools and row-level security
- Scaling out with Azure SQL Database
- Elastic database jobs
- The Elastic Jobs tutorial describes the process of creating, configuring, and managing elastic jobs.

Row-level security

Row-level security is useful for enforcing tenant-level isolation, when you use shared tables.

More information:

- Video overview
- Documentation
- Multi-tenant applications with elastic database tools and row-level security

Key management

The Always Encrypted feature provides the end-to-end encryption of your databases. If your tenants require they supply their own encryption keys, consider deploying separate databases for each tenant and consider enabling the Always Encrypted feature.

More information:

- Always Encrypted

Next steps

See Resources for architects and developers of multitenant solutions.

Related resources

- Data partitioning strategies for Azure SQL Database
- **Case study:** Running 1M databases on Azure SQL for a large SaaS provider: Microsoft Dynamics 365 and Power Platform
- **Sample:** The Wingtip Tickets SaaS application provides three multi-tenant examples of the the same app; each explores a different database tenancy pattern on Azure SQL Database. The first uses a standalone application, per tenant with its own database. The second uses a multi-tenant app with a database, per tenant. The third sample uses a multi-tenant app with sharded multi-tenant databases.
- **Video:** Multitenant design patterns for SaaS applications on Azure SQL Database

Multitenancy and Azure Cosmos DB

11/19/2021 • 8 minutes to read • Edit Online

On this page, we describe some of the features of Azure Cosmos DB that are useful when working with multitenanted systems, and we link to guidance and examples for how to use Azure Cosmos DB in a multitenant solution.

Features of Azure Cosmos DB that support multitenancy

Partitioning

By using partitions with your Cosmos DB containers, you can create containers that are shared across multiple tenants. Typically you use the tenant identifier as a partition key, but you might also consider using multiple partition keys for a single tenant. A well-planned partitioning strategy effectively implements the Sharding pattern. With large containers, Cosmos DB spreads your tenants across multiple physical nodes to achieve a high degree of scale.

More information:

- Partitioning and horizontal scaling in Azure Cosmos DB
- Hierarchical partition keys

Managing request units

Cosmos DB's pricing model is based on the number of *request units* per second that you provision or consume. A request unit is a logical abstraction of the cost of a database operation or query. Typically, you provision a defined number of request units per second for your workload, which is referred to as *throughput*. Cosmos DB provides several options for how you provision throughput. In a multitenant environment, the selection you make affects the performance and price of your Cosmos DB resources.

One tenancy model for Cosmos DB involves deploying separate containers for each tenant within a shared database. Cosmos DB enables you to provision request units for a database, and all of the containers share the request units. If your tenant workloads don't typically overlap, this can provide a useful approach to reduce your operational costs. However, note that this approach is susceptible to the Noisy Neighbor problem because a single tenant's container might consume a disproportionate amount of the shared provisioned request units. To mitigate this after you've identified noisy tenants, you can optionally set provisioned throughput on a specific container. The other containers in the database continue to share their throughput, but the noisy tenant consumes their own dedicated throughput.

Cosmos DB also provides a serverless tier, which is suited for workloads with intermittent or unpredictable traffic. Alternatively, autoscaling enables you to configure policies to specify the scaling of provisioned throughput. In a multitenant solution, you might combine all of these approaches to support different types of tenant.

> **NOTE**
>
> When planning your Cosmos DB configuration, ensure you consider the service quotas and limits.

To monitor and manage the costs that are associated with each tenant, every operation using the Cosmos DB API includes the request units consumed. You can use this information to aggregate and compare the actual request units consumed by each tenant, and you can then identify tenants with different performance characteristics.

More information:

- Provisioned throughput
- Autoscale
- Serverless
- Measuring the RU charge of a request
- Azure Cosmos DB service quotas

Customer-managed keys

Some tenants might require the use of their own encryption keys. Cosmos DB provides a customer-managed key feature. This feature is applied at the level of a Cosmos DB account, so tenants who require their own encryption keys need to be deployed using dedicated Cosmos DB accounts.

More information:

- Configure customer-managed keys for your Azure Cosmos account with Azure Key Vault

Isolation models

When working with a multitenant system that uses Azure Cosmos DB, you need to make a decision about the level of isolation you want to use. Azure Cosmos DB supports several isolation models:

	SHARED CONTAINERS WITH PARTITION KEYS PER TENANT	CONTAINER WITH SHARED THROUGHPUT PER TENANT	CONTAINER WITH DEDICATED THROUGHPUT PER TENANT	DATABASE ACCOUNT PER TENANT
Isolation options	• Share throughput across tenants grouped by container (great for lowering cost on 'spiky' tenants). • Enables easy queries across tenants (containers act as boundary for queries). Mitigate a noisy neighbor blast radius (group tenants by container).	• Share throughput across tenants that are grouped by database (which is great for lowering costs on 'spiky' tenants). • Easy management of tenants (drop the container when the tenant leaves). • Mitigate noisy neighbor blast radius (group tenants by database).	• Independent throughput options (the dedicated throughput eliminates noisy neighbors). • Group tenants within database account(s), based on regional needs.	• Independent geo-replication knobs. • Multiple throughput options (the dedicated throughput eliminates noisy neighbors).
Throughput requirements	>0 RUs per tenant	>100 RUs per tenant	>400 RUs per tenant	>400 RUs per tenant
Example use case	B2C apps	Standard offer for B2B apps	Premium offer for B2B apps	Premium offer for B2B apps

Shared container with partition keys per tenant

When you use a single container for multiple tenants, you can make use of Cosmos DB's partitioning support. By using separate partition keys for each tenant, you can easily query the data for a single tenant. You can also query across multiple tenants, even if they are in separate partitions. However, cross-partition queries have a higher request unit (RU) cost than single-partition queries.

This approach tends to work well when the amount of data stored for each tenant is small. It can be a good choice for building a pricing model that includes a free tier, and for business-to-consumer (B2C) solutions. In general, by using shared containers, you achieve the highest density of tenants and therefore the lowest price per tenant.

It's important to consider the throughput of your container. All of the tenants will share the container's throughput, so the Noisy Neighbor problem can cause performance challenges if your tenants have high or overlapping workloads. This problem can sometimes be mitigated by grouping subsets of tenants into different containers, and by ensuring that each tenant group has compatible usage patterns. Alternatively, you can consider a hybrid multi- and single-tenant model, where smaller tenants are grouped into shared partitioned containers, and large customers have dedicated containers.

It's also important to consider the amount of data you store in each logical partition. Azure Cosmos DB allows each logical partition to grow to up to 20 GB. If you have a single tenant that needs to store more than 20 GB of data, consider spreading the data across multiple logical partitions. For example, instead of having a single partition key of `Contoso`, you might *salt* the partition keys by creating multiple partition keys for a tenant, such as `Contoso1`, `Contoso2`, and so forth. When you query the data for a tenant, you can use the `WHERE IN` clause to match all of the partition keys. Hierarchical partition keys can also be used to support large tenants.

Consider the operational aspects of your solution, and the different phases of the tenant lifecycle. For example, when a tenant moves to a dedicated pricing tier, you will likely need to move the data to a different container. When a tenant is deprovisioned, you need to run a delete query on the container to remove the data, and for large tenants, this query might consume a significant amount of throughput while it executes.

Container per tenant

You can provision dedicated containers for each tenant. This can work well when the data you store for your tenant can be combined into a single container.

When using a container for each tenant, you can consider sharing throughput with other tenants by provisioning throughput at the database level. Consider the restrictions and limits around the minimum number of request units for your database and the maximum number of containers in the database. Also, consider whether your tenants require a guaranteed level of performance, and whether they're susceptible to the Noisy Neighbor problem. If necessary, plan to group tenants into different databases that are based on workload patterns.

Alternatively, you can provision dedicated throughput for each container. This works well for larger tenants, and for tenants that are at risk of the Noisy Neighbor problem. However, the baseline throughput for each tenant is higher, so consider the minimum requirements and cost implications of this model.

Lifecycle management is generally simpler when containers are dedicated to tenants. You can easily move tenants between shared and dedicated throughput models, and when deprovisioning a tenant, you can quickly delete the container.

Database account per tenant

Cosmos DB enables you to provision separate database accounts for each tenant, which provides the highest level of isolation, but the lowest density. A single database account is dedicated to a tenant, which means they are not subject to the noisy neighbor problem. You can also configure the location of the database account according to the tenant's requirements, and you can tune the configuration of Cosmos DB features, such as geo-replication and customer-managed encryption keys, to suit each tenant's requirements. When using a dedicated

Cosmos DB account per tenant, consider the maximum number of Cosmos DB accounts per Azure subscription.

If you allow tenants to migrate from a shared account to a dedicated Cosmos DB account, consider the migration approach you'll use to move a tenant's data between the old and new accounts.

Hybrid approaches

You can consider a combination of the above approaches to suit different tenants' requirements and your pricing model. For example:

- Provision all free trial customers within a shared container, and use the tenant ID or a synthetic key partition key.
- Offer a paid *Bronze* tier that deploys a dedicated container per tenant, but with shared throughput on a database.
- Offer a higher *Silver* tier that provisions dedicated throughput for the tenant's container.
- Offer the highest *Gold* tier, and provide a dedicated database account for the tenant, which also allows tenants to select the geography for their deployment.

Next steps

Review Resources for architects and developers of multitenant solutions.

Related resources

- Azure Cosmos DB and multitenant systems: A blog post that discusses how to build a multitenant system that uses Azure Cosmos DB.
- Multitenant applications with Azure Cosmos DB (video)
- Building a multitenant SaaS with Azure Cosmos DB and Azure (video): A real-world case study of how Whally, a multitenant SaaS startup, built a modern platform from scratch on Azure Cosmos DB and Azure. Whally shows the design and implementation decisions they made that relate to partitioning, data modeling, secure multitenancy, performance, real-time streaming from change feed to SignalR, and more, all using ASP.NET Core on Azure App Services.

Resources for architects and developers of multitenant solutions

11/19/2021 • 7 minutes to read • Edit Online

Architectures for multitenant applications

The following articles provide examples of multitenant architectures on Azure.

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Multitenant SaaS on Azure	Reference architecture for a multitenant SaaS scenario on Azure, which is deployed in multiple regions	Web
Use Application Gateway Ingress Controller with a multi-tenant Azure Kubernetes Service	Example for implementing multitenancy with AKS and AGIC	Kubernetes
Serverless batch processing with Durable Functions in Azure Container Instances	Use cases include multitenant scenarios, where some tenants need large computing power, while other tenants have small computing requirements	Containers
All multitenant architectures	Lists all the architectures that include multitenancy	Multiple

Cloud design patterns

The following cloud design patterns are frequently used in multitenant architectures.

PATTERN	SUMMARY
Deployment Stamps pattern	Deploy multiple independent copies (scale units) of application components, including data stores.
Federated Identity	Delegate authentication to an external identity provider.
Gatekeeper	Protect applications and services, by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes, in order to smooth intermittent heavy loads.
Sharding	Divide a data store into a set of horizontal partitions or shards.

PATTERN	SUMMARY
Throttling	Control the consumption of resources that are used by an instance of an application, an individual tenant, or an entire service.

Antipatterns

Consider the Noisy Neighbor antipattern, in which the activity of one tenant can have a negative impact on another tenant's use of the system.

Microsoft Azure Well-Architected Framework

While the entirety of the Azure Well-Architected Framework is important for all solutions, pay special attention to the Resiliency pillar. The nature of cloud hosting leads to applications that are often multitenant, use shared platform services, compete for resources and bandwidth, communicate over the internet, and run on commodity hardware. This increases the likelihood that both transient and more permanent faults will arise.

Multitenant architectural guidance

- Accelerate and De-Risk Your Journey to SaaS (video): This video provides guidance for transitioning to the "software as a service" (SaaS) delivery model - whether you're starting by lifting-and-shifting an existing solution from on-premises to Azure, considering a multitenant architecture, or looking to modernize an existing SaaS web application.

Resources for Azure services

Governance and compliance

- Organizing and managing multiple Azure subscriptions: It's important to consider how you manage your Azure subscriptions, as well as how you allocate tenant resources to subscriptions.
- Cross-tenant management experiences: As a service provider, you can use Azure Lighthouse to manage resources, for multiple customers from within your own Azure Active Directory (Azure AD) tenant. Many tasks and services can be performed across managed tenants, by using Azure delegated resource management.
- Azure Managed Applications: In a managed application, the resources are deployed to a resource group that's managed by the publisher of the app. The resource group is present in the consumer's subscription, but an identity in the publisher's tenant has access to the resource group.

Compute

- Best practices for cluster isolation in Azure Kubernetes Service: AKS provides flexibility in how you can run multitenant clusters and can isolate resources. To maximize your investment in Kubernetes, you must first understand and implement AKS multitenancy and isolation features. This best practices article focuses on isolation for cluster operators.
- Best practices for cluster security and upgrades in Azure Kubernetes Service: As you manage clusters in Azure Kubernetes Service (AKS), workload and data security is a key consideration. When you run multitenant clusters using logical isolation, you especially need to secure resource and workload access.

Storage and data

- Azure Cosmos DB and multitenant systems: A blog post discussing how to build a multitenant system that uses Azure Cosmos DB.
- Azure Cosmos DB hierarchical partition keys (private preview): A blog post announcing the private preview of hierarchical partition keys for the Azure Cosmos DB Core (SQL) API. With hierarchical partition keys, also

known as sub-partitioning, you can now natively partition your container with up to three levels of partition keys. This enables more optimal partitioning strategies for multitenant scenarios or workloads that would otherwise use synthetic partition keys.

- Azure SQL Database multitenant SaaS database tenancy patterns: A set of articles describing various tenancy models that are available for a multitenant SaaS application, using Azure SQL Database.
- Running 1 million databases on Azure SQL for a large SaaS provider: Microsoft Dynamics 365 and Power Platform: A blog post describing how Dynamics 365 team manages databases at scale.
- Design a multitenant database by using Azure Database for PostgreSQL Hyperscale
- Horizontal, vertical, and functional data partitioning: In many large-scale and multitenant solutions, data is divided into partitions that can be managed and accessed separately. Partitioning can improve scalability, reduce contention, and optimize performance. It can also provide a mechanism for dividing data, by the usage pattern and by the tenant.
- Data partitioning strategies by Azure service: This article describes some strategies for partitioning data in various Azure data stores.
- Building multitenant applications with Azure Database for PostgreSQL Hyperscale Citus (video)
- Multitenant applications with Azure Cosmos DB (video)
- Building a multitenant SaaS with Azure Cosmos DB and Azure (video): A real-world case study of how Whally, a multitenant SaaS startup, built a modern platform from scratch on Azure Cosmos DB and Azure. Whally shows the design and implementation decisions they made related to partitioning, data modeling, secure multitenancy, performance, real-time streaming from change feed to SignalR and more, all using ASP.NET Core on Azure App Services.
- Multitenant design patterns for SaaS applications on Azure SQL Database (video)

Messaging

- Azure Event Grid domains: Azure Event Grid domains allow you to manage multitenant eventing architectures, at scale.

Identity

- Tenancy in Azure Active Directory: When developing apps, developers can choose to configure their app to be either single-tenant or multitenant, during app registration, in Azure Active Directory.
- Custom-branded identity solution with Azure AD B2C: Azure Active Directory B2C is a customer identity access management solution that is capable of supporting millions of users and billions of authentications per day.
- Identity management in multitenant applications: This series of articles describes best practices for multitenancy, when using Azure AD for authentication and identity management.
- Build a multi-tenant daemon with the Microsoft identity platform endpoint: This sample application shows how to use the Microsoft identity platform endpoint to access the data of Microsoft business customers in a long-running, non-interactive process. It uses the OAuth2 client credentials grant to acquire an access token, which it then uses to call the Microsoft Graph and access organizational data.
- Authenticate and authorize multitenant apps using Azure Active Directory (Azure AD): Learn how Azure Active Directory enables you to improve the functionality of cloud-native apps in multitenant scenarios.
- Azure Architecture Walkthrough: Building a multi-tenant Azure Architecture for a B2C scenario: a walk through the architecture behind a multi-tenant mobile app with Azure Active Directory B2C and API Management.

Analytics

- Multitenancy solutions with Power BI embedded analytics: When designing a multitenant application that contains Power BI Embedded, you must carefully choose the tenancy model that best fits your needs.

IoT

- Multitenancy in IoT Hub Device Provisioning Service: A multitenant IoT solution will commonly assign tenant

devices, by using a group of IoT hubs that are scattered across regions.

AI/ML

- Design patterns for multitenant SaaS applications and Azure Cognitive Search: This document discusses tenant isolation strategies for multitenant applications that are built with Azure Cognitive Search.

Community Content

Kubernetes

- Three Tenancy Models For Kubernetes: Kubernetes clusters are typically used by several teams in an organization. This article explains three tenancy models for Kubernetes.
- Understanding Kubernetes Multi Tenancy: Kubernetes is not a multi-tenant system out of the box. While it is possible to configure multi-tenancy, this can be challenging. This article explains Kubernetes multi-tenancy types.
- Kubernetes Multi-Tenancy – A Best Practices Guide: Kubernetes multi-tenancy is a topic that more and more organizations are interested in as their Kubernetes usage spreads out. However, since Kubernetes is not a multi-tenant system per se, getting multi-tenancy right comes with some challenges. This article describes these challenges and how to overcome them as well as some useful tools for Kubernetes multi-tenancy.
- Capsule: Kubernetes multi-tenancy made simple: Capsule helps to implement a multi-tenancy and policy-based environment in your Kubernetes cluster. It is not intended to be yet another PaaS, instead, it has been designed as a micro-services-based ecosystem with the minimalist approach, leveraging only on upstream Kubernetes.
- Loft: Add Multi-Tenancy To Your Clusters: Loft provides lightweight Kubernetes extensions for multi-tenancy.

Azure for AWS professionals

11/19/2021 • 2 minutes to read • Edit Online

This article helps Amazon Web Services (AWS) experts understand the basics of Microsoft Azure accounts, platform, and services. It also covers key similarities and differences between the AWS and Azure platforms.

You'll learn:

- How accounts and resources are organized in Azure.
- How available solutions are structured in Azure.
- How the major Azure services differ from AWS services.

Azure and AWS built their capabilities independently over time, so that each has important implementation and design differences.

Azure for AWS overview

Like AWS, Microsoft Azure is built around a core set of compute, storage, database, and networking services. In many cases, both platforms offer a basic equivalence between the products and services they offer. Both AWS and Azure allow you to build highly available solutions based on Windows or Linux hosts. So, if your preference for development is using Linux distributions and OSS technologies, both platforms are capable of doing the job.

While the capabilities of both platforms are similar, the resources that provide those capabilities are often organized differently. Exact one-to-one relationships between the services required to build a solution are not always clear. In other cases, a particular service might be offered on one platform, but not the other. See charts of comparable Azure and AWS services.

Services

For a listing of how services map between the platforms, see AWS to Azure services comparison.

Not all Azure products and services are available in all regions. Consult the Products by Region page for more details. You can find the uptime guarantees and downtime credit policies for each Azure product or service on the Service Level Agreements page.

Components

A number of core components on Azure and AWS have similar functionality. To review the differences, visit the component page for the topic you're interested in:

- Accounts
- Compute
- Databases
- Messaging
- Networking
- Regions and Zones
- Resources
- Security & Identity
- Storage

Azure and AWS accounts and subscriptions

11/19/2021 • 2 minutes to read • Edit Online

Azure services can be purchased using several pricing options, depending on your organization's size and needs. See the pricing overview page for details.

Azure subscriptions are a grouping of resources with an assigned owner responsible for billing and permissions management. Unlike AWS, where any resources created under the AWS account are tied to that account, subscriptions exist independently of their owner accounts, and can be reassigned to new owners as needed.

Comparison of structure and ownership of AWS accounts and Azure subscriptions

An Azure account represents a billing relationship and Azure subscriptions help you organize access to Azure resources. Account Administrator, Service Administrator, and Co-Administrator are the three classic subscription administrator roles in Azure:

- **Account Administrator**. The subscription owner and the billing owner for the resources used in the subscription. The account administrator can only be changed by transferring ownership of the subscription. Only one Account administrator is assigned per Azure Account.
- **Service Administrator**. This user has rights to create and manage resources in the subscription, but is not responsible for billing. By default, for a new subscription, the Account Administrator is also the Service Administrator. The account administrator can assign a separate user to the service administrator for managing the technical and operational aspects of a subscription. Only one service administrator is assigned per subscription.
- **Co-administrator**. There can be multiple co-administrators assigned to a subscription. Co-administrators have the same access privileges as the Service Administrator, but they cannot change the service administrator.

Below the subscription level user roles and individual permissions can also be assigned to specific resources, similarly to how permissions are granted to IAM users and groups in AWS. In Azure, all user accounts are associated with either a Microsoft Account or Organizational Account (an account managed through an Azure Active Directory).

Like AWS accounts, subscriptions have default service quotas and limits. For a full list of these limits, see Azure subscription and service limits, quotas, and constraints. These limits can be increased up to the maximum by filing a support request in the management portal.

See also

- Classic subscription administrator roles, Azure roles, and Azure AD roles
- How to add or change Azure administrator roles
- How to download your Azure billing invoice and daily usage data

Compute services on Azure and AWS

11/19/2021 • 4 minutes to read • Edit Online

Amazon EC2 Instances and Azure virtual machines

Although AWS instance types and Azure virtual machine sizes are categorized similarly, the RAM, CPU, and storage capabilities differ.

- Amazon EC2 Instance Types
- Sizes for virtual machines in Azure (Windows)
- Sizes for virtual machines in Azure (Linux)

Similar to AWS' per second billing, Azure on-demand VMs are billed per second.

Amazon EBS and Azure Storage for VM disks

Durable data storage for Azure VMs is provided by data disks residing in blob storage. This is similar to how EC2 instances store disk volumes on Elastic Block Store (EBS). Azure temporary storage also provides VMs with the same low-latency temporary read-write storage as EC2 Instance Storage (also called ephemeral storage). Azure Files provides the VMs with the same functionality as Amazon EFS.

Higher performance disk I/O is supported using Azure premium storage. This is similar to the Provisioned IOPS storage options provided by AWS.

AWS Lambda, Azure Functions, Azure Web-Jobs, and Azure Logic Apps

Azure Functions is the primary equivalent of AWS Lambda in providing serverless, on-demand code. However, AWS Lambda functionality also overlaps with other Azure services:

- WebJobs allow you to create scheduled or continuously running background tasks.

Autoscaling, Azure VM scaling, and Azure App Service Autoscale

Autoscaling in Azure is handled by two services:

- Virtual machine scale sets allow you to deploy and manage an identical set of VMs. The number of instances can autoscale based on performance needs.
- App Service Autoscale provides the capability to autoscale Azure App Service solutions.

Container Service

The Azure Kubernetes Service supports Docker containers managed through Kubernetes. See Container runtime configuration for specifics on the hosting environment.

Distributed Systems Platform

Service Fabric is a platform for developing and hosting scalable microservices-based solutions.

Batch Processing

Azure Batch allows you to manage compute-intensive work across a scalable collection of virtual machines.

Service Comparison

Virtual servers

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Compute Cloud (EC2) Instances	Virtual Machines	Virtual servers allow users to deploy, manage, and maintain OS and server software. Instance types provide combinations of CPU/RAM. Users pay for what they use with the flexibility to change sizes.
Batch	Batch	Run large-scale parallel and high-performance computing applications efficiently in the cloud.
Auto Scaling	Virtual Machine Scale Sets	Allows you to automatically change the number of VM instances. You set defined metric and thresholds that determine if the platform adds or removes instances.
VMware Cloud on AWS	Azure VMware Solution	Seamlessly move VMware-based workloads from your datacenter to Azure and integrate your VMware environment with Azure. Keep managing your existing environments with the same VMware tools you already know while you modernize your applications with Azure native services. Azure VMware Solution is a Microsoft service, verified by VMware, that runs on Azure infrastructure.
Parallel Cluster	CycleCloud	Create, manage, operate, and optimize HPC and big compute clusters of any scale

Containers and container orchestrators

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Container Service (ECS) Fargate	Container Instances	Azure Container Instances is the fastest and simplest way to run a container in Azure, without having to provision any virtual machines or adopt a higher-level orchestration service.
Elastic Container Registry	Container Registry	Allows customers to store Docker formatted images. Used to create all types of container deployments on Azure.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Kubernetes Service (EKS)	Kubernetes Service (AKS)	Deploy orchestrated containerized applications with Kubernetes. Simplify monitoring and cluster management through auto upgrades and a built-in operations console. See AKS solution journey.
App Mesh	Service Fabric Mesh	Fully managed service that enables developers to deploy microservices applications without managing virtual machines, storage, or networking.

Container architectures

- Baseline architecture on Azure Kubernetes Service (AKS)

  - 07/20/2020
  - 37 min read

  Deploy a baseline infrastructure that deploys an AKS cluster with focus on security.

- Microservices architecture on Azure Kubernetes Service (AKS)

  - 5/07/2020
  - 17 min read

  Deploy a microservices architecture on Azure Kubernetes Service (AKS)

- CI/CD pipeline for container-based workloads

  - 7/05/2018
  - 7 min read

  Build a DevOps pipeline for a Node.js web app with Jenkins, Azure Container Registry, Azure Kubernetes Service, Cosmos DB, and Grafana.

view all

Serverless

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Lambda	Functions	Integrate systems and run backend processes in response to events or schedules without provisioning or managing servers.

Serverless architectures

Social App for Mobile and Web with Authentication

- 12/16/2019
- 3 min read

View a detailed, step-by-step diagram depicting the build process and implementation of the mobile client app architecture that offers social image sharing with a companion web app and authentication abilities, even while offline.

-

HIPAA and HITRUST compliant health data AI

- 12/16/2019
- 2 min read

Manage HIPAA and HITRUST compliant health data and medical records with the highest level of built-in security.

-

Cross Cloud Scaling Architecture

- 12/16/2019
- 1 min read

Learn how to improve cross cloud scalability with solution architecture that includes Azure Stack. A step-by-step flowchart details instructions for implementation.

See also

- Create a Linux VM on Azure using the portal
- Azure Reference Architecture: Running a Linux VM on Azure
- Get started with Node.js web apps in Azure App Service
- Azure Reference Architecture: Basic web application
- Create your first Azure Function

Relational database technologies on Azure and AWS

11/19/2021 • 2 minutes to read • Edit Online

RDS and Azure relational database services

Azure provides several different relational database services that are the equivalent of AWS' Relational Database Service (RDS). These include:

- SQL Database
- Azure Database for MySQL
- Azure Database for PostgreSQL
- Azure Database for MariaDB

Other database engines such as SQL Server, Oracle, and MySQL can be deployed using Azure VM Instances.

Costs for AWS RDS are determined by the amount of hardware resources that your instance uses, like CPU, RAM, storage, and network bandwidth. In the Azure database services, cost depends on your database size, concurrent connections, and throughput levels.

See also

- Azure SQL Database Tutorials
- Azure SQL Managed Instance
- Configure geo-replication for Azure SQL Database with the Azure portal
- Introduction to Cosmos DB: A NoSQL JSON Database
- How to use Azure Table storage from Node.js

Analytics and big data

Azure provides a package of products and services designed to capture, organize, analyze, and visualize large amounts of data consisting of the following services:

- HDInsight: managed Apache distribution that includes Hadoop, Spark, Storm, or HBase.
- Data Factory: provides data orchestration and data pipeline functionality.
- Azure Synapse Analytics: an enterprise analytics service that accelerates time to insight, across data warehouses and big data systems.
- Azure Databricks: a unified analytics platform for data analysts, data engineers, data scientists, and machine learning engineers.
- Data Lake Store: analytics service that brings together enterprise data warehousing and big data analytics. Query data on your terms, using either serverless or dedicated resources—at scale.
- Machine Learning: used to build and apply predictive analytics on data.
- Stream Analytics: real-time data analysis.
- Data Lake Analytics: large-scale analytics service optimized to work with Data Lake Store

- Power BI: a business analytics service that provides the capabilities to create rich interactive data visualizations.

Service comparison

TYPE	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Relational database	RDS	SQL Database Database for MySQL Database for PostgreSQL Database for MariaDB	Managed relational database services in which resiliency, scale and maintenance are primarily handled by the Azure platform.
Serverless relational database	Amazon Aurora Serverless	Azure SQL Database serverless Serverless SQL pool in Azure Synapse Analytics	Database offerings that automatically scales compute based on the workload demand. You're billed per second for the actual compute used (Azure SQL)/data that's processed by your queries (Azure Synapse Analytics Serverless).
NoSQL/ Document	DynamoDB SimpleDB Amazon DocumentDB	Cosmos DB	Cosmos DB is a globally distributed, multi-model database that natively supports multiple data models including key-value pairs, documents, graphs and columnar.
Caching	ElastiCache	Cache for Redis	An in-memory–based, distributed caching service that provides a high-performance store typically used to offload nontransactional work from a database.
Database migration	Database Migration Service	Database Migration Service	A service that executes the migration of database schema and data from one database format to a specific database technology in the cloud.

Database architectures

-

Gaming using Cosmos DB

- 12/16/2019
- 1 min read

Elastically scale your database to accommodate unpredictable bursts of traffic and deliver low-latency

multi-player experiences on a global scale.

- Oracle Database Migration to Azure

  - 12/16/2019
  - 2 min read

  Oracle DB migrations can be accomplished in multiple ways. This architecture covers one of these options wherein Oracle Active Data Guard is used to migrate the Database.

- Retail and e-commerce using Azure MySQL

  - 12/16/2019
  - 1 min read

  Build secure and scalable e-commerce solutions that meet the demands of both customers and business using Azure Database for MySQL.

view all

See also

- Azure AI Gallery
- Cloud-scale analytics
- Big data architecture style
- Azure Data Lake and Azure HDInsight blog

Messaging services on Azure and AWS

11/19/2021 • 2 minutes to read • Edit Online

Simple Email Service

AWS provides the Simple Email Service (SES) for sending notification, transactional, or marketing emails. In Azure, third-party solutions, like SendGrid, provide email services that can be incorporated into solutions to cater for various use cases.

Simple Queueing Service

AWS Simple Queueing Service (SQS) provides a messaging system for connecting applications, services, and devices within the AWS platform. Azure has two services that provide similar functionality:

- Queue storage is a cloud messaging service that allows communication between application components within Azure.
- Service Bus is a robust messaging system for connecting applications, services, and devices. By using the related Service Bus relay, Service Bus can also connect to remotely hosted applications and services.

Messaging components

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Simple Queue Service (SQS)	Queue Storage	Provides a managed message queueing service for communicating between decoupled application components.
Simple Notification Service (SNS)	Service Bus	Supports a set of cloud-based, message-oriented middleware technologies, including reliable message queuing and durable publish/subscribe messaging.
Amazon EventBridge	Event Grid	A fully managed event routing service that allows for uniform event consumption using a publish/subscribe model.
Amazon Kinesis	Event Hub	A fully managed, real-time data ingestion service. Stream millions of events per second, from any source, to build dynamic data pipelines and to immediately respond to business challenges.
Amazon MQ	Service Bus	Service Bus Premium is fully compliant with the Java/Jakarta EE Java Message Service (JMS) 2.0 API. Service Bus Standard supports the JMS 1.1 subset focused on queues.

Messaging architectures

-

  Anomaly Detector Process

  - 12/16/2019
  - 1 min read

  Learn more about Anomaly Detector with a step-by-step flowchart that details the process. See how anomaly detection models are selected with time-series data.

-

  Scalable web application

  - 10/03/2019
  - 7 min read

  Use the proven practices in this reference architecture to improve scalability and performance in an Azure App Service web application..

-

  Enterprise integration using queues and events

  - 12/03/2018
  - 5 min read

  Recommended architecture for implementing an enterprise integration pattern with Azure Logic Apps, Azure API Management, Azure Service Bus, and Azure Event Grid.

-

  Ops automation using Event Grid

  - 12/16/2019
  - 1 min read

  Event Grid allows you to speed automation and simplify policy enforcement. For example, Event Grid can notify Azure Automation when a virtual machine is created, or a SQL Database is spun up. These events can be used to automatically check that service configurations are compliant, put metadata into operations tools, tag virtual machines, or file work items.

Networking on Azure and AWS

11/19/2021 • 5 minutes to read • Edit Online

Elastic Load Balancing, Azure Load Balancer, and Azure Application Gateway

The Azure equivalent of the Elastic Load Balancing services are:

- Load Balancer: Provides the same network layer 4 capabilities as the AWS Network Load Balancer and Classic Load Balancer, allowing you to distribute traffic for multiple VMs at the network level. It also provides a failover capability.
- Application Gateway: Offers application-level rule-based routing comparable to the AWS Application Load Balancer.

Route 53, Azure DNS, and Azure Traffic Manager

In AWS, Route 53 provides both DNS name management and DNS-level traffic routing and failover services. In Azure this is handled through two services:

- Azure DNS provides domain and DNS management.
- Traffic Manager provides DNS level traffic routing, load balancing, and failover capabilities.

Direct connect and Azure ExpressRoute

Azure provides similar site-to-site dedicated connections through its ExpressRoute service. ExpressRoute allows you to connect your local network directly to Azure resources using a dedicated private network connection. Azure also offers more conventional site-to-site VPN connections at a lower cost.

Route tables

AWS provides route tables that contain routes to direct traffic, from a subnet/gateway subnet to the destination. In Azure, this feature is called user-defined routes.

With user-defined routes, you can create custom or user-defined (static) routes in Azure, to override Azure's default system routes, or to add more routes to a subnet's route table.

Private Link

Similar to AWS PrivateLink, Azure Private Link provides private connectivity from a virtual network to an Azure platform as a service (PaaS) solution, a customer-owned service, or a Microsoft partner service.

VPC Peering, Azure VNet Peering

In AWS, a VPC peering connection is a networking connection between two VPCs, which enables you to route traffic between them using private IPv4 addresses or IPv6 addresses.

Azure virtual network (VNet) peering enables you to seamlessly connect two or more Virtual Networks in Azure. The virtual networks appear as one for connectivity purposes. The traffic between virtual machines in peered virtual networks uses the Microsoft backbone infrastructure. Like traffic between virtual machines in the same network, traffic is routed through Microsoft's private network only.

Content delivery networks - CloudFront and Azure CDN

In AWS, CloudFront provides CDN services, to globally deliver data, videos, applications, and APIs. This is similar to Azure Content Delivery Network (CDN).

Azure CDN is a feature-inclusive content delivery network bundle that is capable of handling most CDN workloads. Customers can choose to use Azure Content Delivery Network Standard from Verizon or Akamai. For a full list of Azure CDN product offerings, see What are the comparisons between Azure CDN product features?

Network service comparison

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Cloud virtual networking	Virtual Private Cloud (VPC)	Virtual Network	Provides an isolated, private environment in the cloud. Users have control over their virtual networking environment, including selection of their own IP address range, creation of subnets, and configuration of route tables and network gateways.
NAT gateways	NAT Gateways	Virtual Network NAT	A service that simplifies outbound-only Internet connectivity for virtual networks. When configured on a subnet, all outbound connectivity uses your specified static public IP addresses. Outbound connectivity is possible without a load balancer or public IP addresses directly attached to virtual machines.
Cross-premises connectivity	VPN Gateway	VPN Gateway	Connects Azure virtual networks to other Azure virtual networks, or customer on-premises networks (Site To Site). Allows end users to connect to Azure services through VPN tunneling (Point To Site).
DNS management	Route 53	DNS	Manage your DNS records using the same credentials and billing and support contract as your other Azure services

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
DNS-based routing	Route 53	Traffic Manager	A service that hosts domain names, plus routes users to Internet applications, connects user requests to datacenters, manages traffic to apps, and improves app availability with automatic failover.
Dedicated network	Direct Connect	ExpressRoute	Establishes a dedicated, private network connection from a location to the cloud provider (not over the Internet).
Load balancing	Network Load Balancer	Load Balancer	Azure Load Balancer load balances traffic at layer 4 (TCP or UDP). Standard Load Balancer also supports cross-region or global load balancing.
Application-level load balancing	Application Load Balancer	Application Gateway	Application Gateway is a layer 7 load balancer. It supports SSL termination, cookie-based session affinity, and round robin for load-balancing traffic.
Route table	Custom Route Tables	User Defined Routes	Custom, or user-defined (static) routes to override default system routes, or to add more routes to a subnet's route table.
Private link	PrivateLink	Azure Private Link	Azure Private Link provides private access to services that are hosted on the Azure platform. This keeps your data on the Microsoft network.
Private PaaS connectivity	VPC endpoints	Private Endpoint	Private Endpoint provides secured, private connectivity to various Azure platform as a service (PaaS) resources, over a backbone Microsoft private network.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Virtual network peering	VPC Peering	VNET Peering	VNet peering is a mechanism that connects two virtual networks (VNets) in the same region through the Azure backbone network. Once peered, the two virtual networks appear as one for all connectivity purposes.
Content delivery networks	Cloud Front	Azure CDN	The Azure Content Delivery Network is designed to send audio, video, apps, photos, and other files to your customers faster and more reliably, using the servers closest to each user. Acceleration Data Transfer provides dynamic site acceleration of non-cacheable, dynamic content that is generated by your web applications.
Network Monitoring	VPC Flow Logs	Azure Network Watcher	Azure Network Watcher allows you to monitor, diagnose, and analyze the traffic in Azure Virtual Network.

Networking architectures

- Deploy highly available NVAs
  - 12/08/2018
  - 7 min read

  Learn how to deploy network virtual appliances for high availability in Azure. This article includes example architectures for ingress, egress, and both.

- Hub-spoke network topology in Azure
  - 9/30/2020
  - 7 min read

  Learn how to implement a hub-spoke topology in Azure, where the hub is a virtual network and the spokes are virtual networks that peer with the hub.

- Implement a secure hybrid network
  - 1/07/2020

- 9 min read

  See a secure hybrid network that extends an on-premises network to Azure with a perimeter network between the on-premises network and an Azure virtual network.

view all

See also

- Create a virtual network using the Azure portal
- Plan and design Azure Virtual Networks
- Azure Network Security Best Practices

Regions and zones on Azure and AWS

11/19/2021 • 3 minutes to read • Edit Online

Failures can vary in the scope of their impact. Some hardware failures, such as a failed disk, may affect a single host machine. A failed network switch could affect a whole server rack. Less common are failures that disrupt a whole datacenter, such as loss of power in a datacenter. Rarely, an entire region could become unavailable.

One of the main ways to make an application resilient is through redundancy. But you need to plan for this redundancy when you design the application. Also, the level of redundancy that you need depends on your business requirements—not every application needs redundancy across regions to guard against a regional outage. In general, a tradeoff exists between greater redundancy and reliability versus higher cost and complexity.

In Azure, a region is divided into two or more Availability Zones. An Availability Zone corresponds with a physically isolated datacenter in the geographic region. Azure has numerous features for providing application redundancy at every level of potential failure, including **availability sets**, **availability zones**, and **paired regions**.

The diagram has three parts. The first part shows VMs in an availability set in a virtual network. The second part shows an availability zone with two availability sets in a virtual network. The third part shows regional pairs with resources in each region.

The following table summarizes each option.

	AVAILABILITY SET	AVAILABILITY ZONE	PAIRED REGION
Scope of failure	Rack	Datacenter	Region
Request routing	Load Balancer	Cross-zone Load Balancer	Traffic Manager
Network latency	Very low	Low	Mid to high
Virtual networking	VNet	VNet	Cross-region VNet peering

Availability sets

To protect against localized hardware failures, such as a disk or network switch failing, deploy two or more VMs in an availability set. An availability set consists of two or more *fault domains* that share a common power source and network switch. VMs in an availability set are distributed across the fault domains, so if a hardware failure affects one fault domain, network traffic can still be routed to the VMs in the other fault domains. For more information about Availability Sets, see Manage the availability of Windows virtual machines in Azure.

When VM instances are added to availability sets, they are also assigned an update domain. An update domain is a group of VMs that are set for planned maintenance events at the same time. Distributing VMs across multiple update domains ensures that planned update and patching events affect only a subset of these VMs at any given time.

Availability sets should be organized by the instance's role in your application to ensure one instance in each role is operational. For example, in a three-tier web application, create separate availability sets for the front-end,

application, and data tiers.

Availability zones

An Availability Zone is a physically separate zone within an Azure region. Each Availability Zone has a distinct power source, network, and cooling. Deploying VMs across availability zones helps to protect an application against datacenter-wide failures.

Paired regions

To protect an application against a regional outage, you can deploy the application across multiple regions, using Azure Traffic Manager to distribute internet traffic to the different regions. Each Azure region is paired with another region. Together, these form a regional pair. With the exception of Brazil South, regional pairs are located within the same geography in order to meet data residency requirements for tax and law enforcement jurisdiction purposes.

Unlike Availability Zones, which are physically separate datacenters but may be in relatively nearby geographic areas, paired regions are typically separated by at least 300 miles. This design ensures that large-scale disasters only affect one of the regions in the pair. Neighboring pairs can be set to sync database and storage service data, and are configured so that platform updates are rolled out to only one region in the pair at a time.

Azure geo-redundant storage is automatically backed up to the appropriate paired region. For all other resources, creating a fully redundant solution using paired regions means creating a full copy of your solution in both regions.

See also

- Regions for virtual machines in Azure
- Availability options for virtual machines in Azure
- High availability for Azure applications
- Failure and disaster recovery for Azure applications
- Planned maintenance for Linux virtual machines in Azure

Resource management on Azure and AWS

11/19/2021 • 2 minutes to read • Edit Online

The term "resource" in Azure is used in the same way as in AWS, meaning any compute instance, storage object, networking device, or other entity you can create or configure within the platform.

Azure resources are deployed and managed using one of two models: Azure Resource Manager, or the older Azure classic deployment model. Any new resources are created using the Resource Manager model.

Resource groups

Both Azure and AWS have entities called "resource groups" that organize resources such as VMs, storage, and virtual networking devices. However, Azure resource groups are not directly comparable to AWS resource groups.

While AWS allows a resource to be tagged into multiple resource groups, an Azure resource is always associated with one resource group. A resource created in one resource group can be moved to another group, but can only be in one resource group at a time. Resource groups are the fundamental grouping used by Azure Resource Manager.

Resources can also be organized using tags. Tags are key-value pairs that allow you to group resources across your subscription irrespective of resource group membership.

Management interfaces

Azure offers several ways to manage your resources:

- Web interface. Like the AWS Dashboard, the Azure portal provides a full web-based management interface for Azure resources.
- REST API. The Azure Resource Manager REST API provides programmatic access to most of the features available in the Azure portal.
- Command Line. The Azure CLI provides a command-line interface capable of creating and managing Azure resources. The Azure CLI is available for Windows, Linux, and Mac OS.
- PowerShell. The Azure modules for PowerShell allow you to execute automated management tasks using a script. PowerShell is available for Windows, Linux, and Mac OS.
- Templates. Azure Resource Manager templates provide similar JSON template-based resource management capabilities to the AWS CloudFormation service.

In each of these interfaces, the resource group is central to how Azure resources get created, deployed, or modified. This is similar to the role a "stack" plays in grouping AWS resources during CloudFormation deployments.

The syntax and structure of these interfaces are different from their AWS equivalents, but they provide comparable capabilities. In addition, many third-party management tools used on AWS, like Hashicorp's Terraform and Netflix Spinnaker, are also available on Azure.

See also

- Azure resource group guidelines

Multi-cloud security and identity with Azure and Amazon Web Services (AWS)

11/19/2021 • 4 minutes to read • Edit Online

Many organizations are finding themselves with a de facto multi-cloud strategy, even if that wasn't their deliberate strategic intention. In a multi-cloud environment, it's critical to ensure consistent security and identity experiences to avoid increased friction for developers, business initiatives and increased organizational risk from cyberattacks taking advantage of security gaps.

Driving security and identity consistency across clouds should include:

- Multi-cloud identity integration
- Strong authentication and explicit trust validation
- Cloud Platform Security (multi-cloud)
- Privilege Identity Management (Azure)
- Consistent end-to-end identity management

Multi-cloud identity integration

Customers using both Azure and AWS cloud platforms benefit from consolidating identity services between these two clouds using Azure Active Directory (Azure AD) and Single Sign-on (SSO) services. This model allows for a consolidated identity plane through which access to services in both clouds can be consistently accessed and governed.

This approach allows for the rich role-based access controls in Azure Active Directory to be enabled across the Identity & Access Management (IAM) services in AWS using rules to associate the user.userprincipalname and user.assignrole attributes from Azure AD into IAM permissions. This approach reduces the number of unique identities users and administrators are required to maintain across both clouds including a consolidation of the identity per account design that AWS employs. The AWS IAM solution allows for and specifically identifies Azure Active Directory as a federation and authentication source for their customers.

A complete walk-through of this integration can be found in the Tutorial: Azure Active Directory single sign-on (SSO) integration with Amazon Web Services (AWS).

Strong authentication and explicit trust validation

Because many customers continue to support a hybrid identity model for Active Directory services, it's increasingly important for security engineering teams to implement strong authentication solutions and block legacy authentication methods associated primarily with on-premises and legacy Microsoft technologies.

A combination of multi-factor authentication (MFA) and conditional access (CA) policies enable enhanced security for common authentication scenarios for end users in your organization. While MFA itself provides an increase level of security to confirm authentications, additional controls can be applied using CA controls to block legacy authentication to both Azure and AWS cloud environments. Strong authentication using only modern authentication clients is only possible with the combination of MFA and CA policies.

Cloud Platform Security (multi-cloud)

Once a common identity has been established in your multi-cloud environment, the Cloud Platform Security (CPS) service of Microsoft Defender for Cloud Apps can be used to discover, monitor, assess, and protect those

services. Using the Cloud Discovery dashboard, security operations personnel can review the apps and resources being used across AWS and Azure cloud platforms. Once services are reviewed and sanctioned for use, the services can then be managed as enterprise applications in Azure Active Directory to enable SAML, password-based, and linked Single Sign-On mode for the convenience of users.

CPS also provides for the ability to assess the cloud platforms connected for misconfigurations and compliance using vendor specific recommended security and configuration controls. This design enables organizations to maintain a single consolidated view of all cloud platform services and their compliance status.

Additionally, CPS provides access and session control policies to prevent and protect your environment from risky endpoints or users when data exfiltration or malicious files are introduced into those platforms.

Privileged Identity Management (Azure)

To limit and control access for your highest privileged accounts in Azure AD, Privileged Identity Management (PIM) can be enabled to provide just-in-time access to services for Azure cloud services. Once deployed, PIM can be used to control and limit access using the assignment model for roles, eliminate persistent access for these privileged accounts, and provide additional discover and monitoring of users with these account types.

When combined with Microsoft Sentinel, workbooks and playbooks can be established to monitor and raise alerts to your security operations center personnel when there is lateral movement of accounts that have been compromised.

Consistent end-to-end identity management

Ensure that all processes include an end-to-end view of all clouds as well as on-premises systems and that security and identity personnel are trained on these processes.

Using a single identity across Azure AD, AWS Accounts and on-premises services enable this end-to-end strategy and allows for greater security and protection of accounts for privileged and non-privileged accounts. Customers who are currently looking to reduce the burden of maintaining multiple identities in their multi-cloud strategy adopt Azure AD to provide consistent and strong control, auditing, and detection of anomalies and abuse of identities in their environment.

Continued growth of new capabilities across the Azure AD ecosystem helps you stay ahead of threats to your environment as a result of using identities as a common control plane in your multi-cloud environments.

See also

- Azure Active Directory B2B: enables access to your corporate applications from partner-managed identities.
- Azure Active Directory B2C: service offering support for single sign-on and user management for consumer-facing applications.
- Azure Active Directory Domain Services: hosted domain controller service, allowing Active Directory compatible domain join and user management functionality.
- Getting started with Microsoft Azure security
- Azure Identity Management and access control security best practices

Compare storage on Azure and AWS

11/19/2021 • 3 minutes to read • Edit Online

S3/EBS/EFS and Azure Storage

In the AWS platform, cloud storage is primarily broken down into three services:

- **Simple Storage Service (S3)**. Basic object storage that makes data available through an Internet accessible API.
- **Elastic Block Storage (EBS)**. Block level storage intended for access by a single VM.
- **Elastic File System (EFS)**. File storage meant for use as shared storage for up to thousands of EC2 instances.

In Azure Storage, subscription-bound storage accounts allow you to create and manage the following storage services:

- Blob storage stores any type of text or binary data, such as a document, media file, or application installer. You can set Blob storage for private access or share contents publicly to the Internet. Blob storage serves the same purpose as both AWS S3 and EBS.
- Table storage stores structured datasets. Table storage is a NoSQL key-attribute data store that allows for rapid development and fast access to large quantities of data. Similar to AWS' SimpleDB and DynamoDB services.
- Queue storage provides messaging for workflow processing and for communication between components of cloud services.
- File storage offers shared storage for legacy applications using the standard server message block (SMB) protocol. File storage is used in a similar manner to EFS in the AWS platform.

Glacier and Azure Storage

Azure Archive Blob Storage is comparable to AWS Glacier storage service. It is intended for rarely accessed data that is stored for at least 180 days and can tolerate several hours of retrieval latency.

For data that is infrequently accessed but must be available immediately when accessed, Azure Cool Blob Storage tier provides cheaper storage than standard blob storage. This storage tier is comparable to AWS S3 - Infrequent Access storage service.

Storage comparison

Object storage

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Simple Storage Services (S3)	Blob storage	Object storage service, for use cases including cloud applications, content distribution, backup, archiving, disaster recovery, and big data analytics.

Virtual server disks

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Block Store (EBS)	managed disks	SSD storage optimized for I/O intensive read/write operations. For use as high-performance Azure virtual machine storage.

Shared files

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic File System	Files	Provides a simple interface to create and configure file systems quickly, and share common files. Can be used with traditional protocols that access files over a network.

Archiving and backup

AWS SERVICE	AZURE SERVICE	DESCRIPTION
S3 Infrequent Access (IA)	Storage cool tier	Cool storage is a lower-cost tier for storing data that is infrequently accessed and long-lived.
S3 Glacier, Deep Archive	Storage archive access tier	Archive storage has the lowest storage cost and higher data retrieval costs compared to hot and cool storage.
Backup	Backup	Back up and recover files and folders from the cloud, and provide offsite protection against data loss.

Hybrid storage

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Storage Gateway	StorSimple	Integrates on-premises IT environments with cloud storage. Automates data management and storage, plus supports disaster recovery.
DataSync	File Sync	Azure Files can be deployed in two main ways: by directly mounting the serverless Azure file shares or by caching Azure file shares on-premises using Azure File Sync.

Bulk data transfer

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Import/Export Disk	Import/Export	A data transport solution that uses secure disks and appliances to transfer large amounts of data. Also offers data protection during transit.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Import/Export Snowball, Snowball Edge, Snowmobile	Data Box	Petabyte- to exabyte-scale data transport solution that uses secure data storage devices to transfer large amounts of data to and from Azure.

Storage architectures

- HIPAA and HITRUST compliant health data AI

  - 12/16/2019
  - 2 min read

  Manage HIPAA and HITRUST compliant health data and medical records with the highest level of built-in security.

- HPC Media Rendering

  - 11/04/2020
  - 2 min read

  Optimize the media rendering process with a step-by-step HPC solution architecture from Azure that combines Azure CycleCloud and HPC Cache.

- Medical Data Storage Solutions

  - 12/16/2019
  - 2 min read

  Store healthcare data effectively and affordably with cloud-based solutions from Azure. Manage medical records with the highest level of built-in security.

view all

See also

- Microsoft Azure Storage Performance and Scalability Checklist
- Azure Storage security guide
- Best practices for using content delivery networks (CDNs)

AWS to Azure services comparison

11/19/2021 • 28 minutes to read • Edit Online

This article helps you understand how Microsoft Azure services compare to Amazon Web Services (AWS). Whether you are planning a multicloud solution with Azure and AWS, or migrating to Azure, you can compare the IT capabilities of Azure and AWS services in all categories.

This article compares services that are roughly comparable. Not every AWS service or Azure service is listed, and not every matched service has exact feature-for-feature parity.

Azure and AWS for multicloud solutions

As the leading public cloud platforms, Azure and AWS each offer a broad and deep set of capabilities with global coverage. Yet many organizations choose to use both platforms together for greater choice and flexibility, as well as to spread their risk and dependencies with a multicloud approach. Consulting companies and software vendors might also build on and use both Azure and AWS, as these platforms represent most of the cloud market demand.

For an overview of Azure for AWS users, see Introduction to Azure for AWS professionals.

Marketplace

AWS SERVICE	AZURE SERVICE	DESCRIPTION
AWS Marketplace	Azure Marketplace	Easy-to-deploy and automatically configured third-party applications, including single virtual machine or multiple virtual machine solutions.

AI and machine learning

AWS SERVICE	AZURE SERVICE	DESCRIPTION
SageMaker	Machine Learning	A cloud service to train, deploy, automate, and manage machine learning models.
Alexa Skills Kit	Bot Framework	Build and connect intelligent bots that interact with your users using text/SMS, Skype, Teams, Slack, Microsoft 365 mail, Twitter, and other popular services.
Lex	Speech Services	API capable of converting speech to text, understanding intent, and converting text back to speech for natural responsiveness.
Lex	Language Understanding (LUIS)	Allows your applications to understand user commands contextually.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Polly, Transcribe	Speech Services	Enables both Speech to Text, and Text into Speech capabilities.
Rekognition	Cognitive Services	Computer Vision: Extract information from images to categorize and process visual data. Face: Detect, identify, and analyze faces and facial expressions in photos.
Skills Kit	Virtual Assistant	The Virtual Assistant Template brings together a number of best practices we've identified through the building of conversational experiences and automates integration of components that we've found to be highly beneficial to Bot Framework developers.

AI and machine learning architectures

- Image classification on Azure
  - 7/05/2018
  - 4 min read

  Learn how to build image processing into your applications by using Azure services such as the Computer Vision API and Azure Functions.
- Predictive Marketing with Machine Learning
  - 12/16/2019
  - 2 min read

  Learn how to build a machine-learning model with Microsoft R Server on Azure HDInsight Spark clusters to recommend actions to maximize the purchase rate.
- Scalable personalization on Azure
  - 5/31/2019
  - 6 min read

  Use machine learning to automate content-based personalization for customers.

view all

Big data and analytics

Data warehouse

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Redshift	Synapse Analytics	Cloud-based Enterprise Data Warehouse (EDW) that uses Massively Parallel Processing (MPP) to quickly run complex queries across petabytes of data.
Lake Formation	Data Share	A simple and safe service for sharing big data

Data warehouse architectures

- Modern Data Warehouse Architecture
  - 12/16/2019
  - 2 min read

  Explore a cloud data warehouse that uses big data. Modern data warehouse brings together all your data and scales easily as your data grows.
- Automated enterprise BI
  - 6/03/2020
  - 13 min read

  Automate an extract, load, and transform (ELT) workflow in Azure using Azure Data Factory with Azure Synapse Analytics.

view all

Time series

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Amazon Timestream	Azure Data Explorer Azure Time Series Insights	Fully managed, low latency, and distributed big data analytics platform that runs complex queries across petabytes of data. Highly optimized for log and time series data. Open and scalable end-to-end IoT analytics service. Collect, process, store, query, and visualize data at Internet of Things (IoT) scale--data that's highly contextualized and optimized for time series.

Time series architectures

- IoT analytics with Azure Data Explorer
  - 8/11/2020

- 3 min read

IoT Telemetry Analytics with Azure Data Explorer demonstrates near real-time analytics over fast flowing, high volume, wide variety of streaming data from IoT devices.

-

Azure Data Explorer interactive analytics

- 8/11/2020
- 3 min read

Interactive Analytics with Azure Data Explorer focuses on its integration with the rest of the data platform ecosystem.

Big data processing

AWS SERVICE	AZURE SERVICE	DESCRIPTION
EMR	Azure Data Explorer	Fully managed, low latency, distributed big data analytics platform to run complex queries across petabytes of data.
EMR	Databricks	Apache Spark-based analytics platform.
EMR	HDInsight	Managed Hadoop service. Deploy and manage Hadoop clusters in Azure.
EMR	Data Lake Storage	Massively scalable, secure data lake functionality built on Azure Blob Storage.

Big data architectures

-

Azure data platform end-to-end

- 1/31/2020
- 7 min read

Use Azure services to ingest, process, store, serve, and visualize data from different sources.

-

Campaign Optimization with Azure HDInsight Spark Clusters

- 12/16/2019
- 4 min read

This solution demonstrates how to build and deploy a machine learning model with Microsoft R Server on Azure HDInsight Spark clusters to recommend actions to maximize the purchase rate of leads targeted by a campaign. This solution enables efficient handling of big data on Spark with Microsoft R Server.

-

Big data analytics with Azure Data Explorer

- 8/11/2020
- 3 min read

Big Data Analytics with Azure Data Explorer demonstrates Azure Data Explorer's abilities to cater to volume, velocity, and variety of data, the three V's of big data.

view all

Data orchestration / ETL

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Data Pipeline, Glue	Data Factory	Processes and moves data between different compute and storage services, as well as on-premises data sources at specified intervals. Create, schedule, orchestrate, and manage data pipelines.
Glue	Azure Purview	A unified data governance service that helps you manage and govern your on-premises, multicloud, and software as a service (SaaS) data.
Dynamo DB	Table Storage, Cosmos DB	NoSQL key-value store for rapid development using massive semi-structured datasets.

Analytics and visualization

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Kinesis Analytics	Stream Analytics Azure Data Explorer Data Lake Analytics Data Lake Store	Storage and analysis platforms that create insights from large quantities of data, or data that originates from many sources.
QuickSight	Power BI	Business intelligence tools that build visualizations, perform ad hoc analysis, and develop business insights from data.
CloudSearch	Cognitive Search	Delivers full-text search and related search analytics and capabilities.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Athena	Data Lake Analytics Azure Synapse Analytics	Provides a serverless interactive query service that uses standard SQL for analyzing databases. Azure Synapse Analytics is a limitless analytics service that brings together data integration, enterprise data warehousing, and big data analytics. It gives you the freedom to query data on your terms, using either serverless or dedicated resources at scale.
Elasticsearch Service	Elastic on Azure	Use the Elastic Stack (Elastic, Logstash, and Kibana) to search, analyze, and visualize in real time.

Analytics architectures

- Advanced Analytics Architecture

  - 12/16/2019
  - 2 min read

  Get near real-time data analytics on streaming services. This big data architecture allows you to combine any data at any scale with custom machine learning.

- Automated enterprise BI

  - 6/03/2020
  - 13 min read

  Automate an extract, load, and transform (ELT) workflow in Azure using Azure Data Factory with Azure Synapse Analytics.

- Mass ingestion and analysis of news feeds on Azure

  - 2/01/2019
  - 5 min read

  Create a pipeline for ingesting and analyzing text, images, sentiment, and other data from RSS news feeds using only Azure services, including Azure Cosmos DB and Azure Cognitive Services.

view all

Compute

Virtual servers

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Compute Cloud (EC2) Instances	Virtual Machines	Virtual servers allow users to deploy, manage, and maintain OS and server software. Instance types provide combinations of CPU/RAM. Users pay for what they use with the flexibility to change sizes.
Batch	Batch	Run large-scale parallel and high-performance computing applications efficiently in the cloud.
Auto Scaling	Virtual Machine Scale Sets	Allows you to automatically change the number of VM instances. You set defined metric and thresholds that determine if the platform adds or removes instances.
VMware Cloud on AWS	Azure VMware Solution	Seamlessly move VMware-based workloads from your datacenter to Azure and integrate your VMware environment with Azure. Keep managing your existing environments with the same VMware tools you already know while you modernize your applications with Azure native services. Azure VMware Solution is a Microsoft service, verified by VMware, that runs on Azure infrastructure.
Parallel Cluster	CycleCloud	Create, manage, operate, and optimize HPC and big compute clusters of any scale

Containers and container orchestrators

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Container Service (ECS) Fargate	Container Instances	Azure Container Instances is the fastest and simplest way to run a container in Azure, without having to provision any virtual machines or adopt a higher-level orchestration service.
Elastic Container Registry	Container Registry	Allows customers to store Docker formatted images. Used to create all types of container deployments on Azure.
Elastic Kubernetes Service (EKS)	Kubernetes Service (AKS)	Deploy orchestrated containerized applications with Kubernetes. Simplify monitoring and cluster management through auto upgrades and a built-in operations console. See AKS solution journey.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
App Mesh	Service Fabric Mesh	Fully managed service that enables developers to deploy microservices applications without managing virtual machines, storage, or networking.

Container architectures

- Baseline architecture on Azure Kubernetes Service (AKS)
  - 07/20/2020
  - 37 min read

  Deploy a baseline infrastructure that deploys an AKS cluster with focus on security.
- Microservices architecture on Azure Kubernetes Service (AKS)
  - 5/07/2020
  - 17 min read

  Deploy a microservices architecture on Azure Kubernetes Service (AKS)
- CI/CD pipeline for container-based workloads
  - 7/05/2018
  - 7 min read

  Build a DevOps pipeline for a Node.js web app with Jenkins, Azure Container Registry, Azure Kubernetes Service, Cosmos DB, and Grafana.

view all

Serverless

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Lambda	Functions	Integrate systems and run backend processes in response to events or schedules without provisioning or managing servers.

Serverless architectures

- Social App for Mobile and Web with Authentication
  - 12/16/2019
  - 3 min read

  View a detailed, step-by-step diagram depicting the build process and implementation of the mobile

client app architecture that offers social image sharing with a companion web app and authentication abilities, even while offline.

- HIPAA and HITRUST compliant health data AI

  - 12/16/2019
  - 2 min read

  Manage HIPAA and HITRUST compliant health data and medical records with the highest level of built-in security.

- Cross Cloud Scaling Architecture

  - 12/16/2019
  - 1 min read

  Learn how to improve cross cloud scalability with solution architecture that includes Azure Stack. A step-by-step flowchart details instructions for implementation.

Database

TYPE	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Relational database	RDS	SQL Database Database for MySQL Database for PostgreSQL Database for MariaDB	Managed relational database services in which resiliency, scale and maintenance are primarily handled by the Azure platform.
Serverless relational database	Amazon Aurora Serverless	Azure SQL Database serverless Serverless SQL pool in Azure Synapse Analytics	Database offerings that automatically scales compute based on the workload demand. You're billed per second for the actual compute used (Azure SQL)/data that's processed by your queries (Azure Synapse Analytics Serverless).
NoSQL/ Document	DynamoDB SimpleDB Amazon DocumentDB	Cosmos DB	Cosmos DB is a globally distributed, multi-model database that natively supports multiple data models including key-value pairs, documents, graphs and columnar.

TYPE	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Caching	ElastiCache	Cache for Redis	An in-memory–based, distributed caching service that provides a high-performance store typically used to offload nontransactional work from a database.
Database migration	Database Migration Service	Database Migration Service	A service that executes the migration of database schema and data from one database format to a specific database technology in the cloud.

Database architectures

- Gaming using Cosmos DB

  - 12/16/2019
  - 1 min read

  Elastically scale your database to accommodate unpredictable bursts of traffic and deliver low-latency multi-player experiences on a global scale.

- Oracle Database Migration to Azure

  - 12/16/2019
  - 2 min read

  Oracle DB migrations can be accomplished in multiple ways. This architecture covers one of these options wherein Oracle Active Data Guard is used to migrate the Database.

- Retail and e-commerce using Azure MySQL

  - 12/16/2019
  - 1 min read

  Build secure and scalable e-commerce solutions that meet the demands of both customers and business using Azure Database for MySQL.

view all

DevOps and application monitoring

AWS SERVICE	AZURE SERVICE	DESCRIPTION

AWS SERVICE	AZURE SERVICE	DESCRIPTION
CloudWatch, X-Ray	Monitor	Comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments.
CodeDeploy CodeCommit CodePipeline	DevOps	A cloud service for collaborating on code development.
Developer Tools	Developer Tools	Collection of tools for building, debugging, deploying, diagnosing, and managing multiplatform scalable apps and services.
CodeBuild	DevOps Pipeline Github Actions	Fully managed build service that supports continuous integration and deployment.
Command Line Interface	CLI PowerShell	Built on top of the native REST API across all cloud services, various programming language-specific wrappers provide easier ways to create solutions.
eksctl	az aks	Manage Azure Kubernetes Services.
AWS CloudShell	Azure Cloud Shell	Azure Cloud Shell is an interactive, authenticated, browser-accessible shell for managing Azure resources. It gives you the flexibility to choose the shell experience that best suits the way you work, either Bash or PowerShell.
OpsWorks (Chef-based)	Automation	Configures and operates applications of all shapes and sizes, and provides templates to create and manage a collection of resources.
CloudFormation	Resource Manager Bicep VM extensions Azure Automation	Provides a way for users to automate the manual, long-running, error-prone, and frequently repeated IT tasks.

DevOps architectures

- Container CI/CD using Jenkins and Kubernetes on Azure Kubernetes Service (AKS)
  - 12/16/2019
  - 2 min read

Containers make it easy for you to continuously build and deploy applications. By orchestrating the deployment of those containers using Azure Kubernetes Service (AKS), you can achieve replicable, manageable clusters of containers.

-

  Run a Jenkins server on Azure

  - 11/19/2020
  - 6 min read

  Recommended architecture that shows how to deploy and operate a scalable, enterprise-grade Jenkins server on Azure secured with single sign-on (SSO).

-

  DevOps in a hybrid environment

  - 12/16/2019
  - 3 min read

  The tools provided in Azure allow for the implementation of a DevOps strategy that capably manages both cloud and on-premises environments in tandem.

view all

Internet of things (IoT)

AWS SERVICE	AZURE SERVICE	DESCRIPTION
IoT Core	IoT Hub	A cloud gateway for managing bidirectional communication with billions of IoT devices, securely and at scale.
Greengrass	IoT Edge	Deploy cloud intelligence directly onto IoT devices, catering to on-premises scenarios.
Kinesis Firehose, Kinesis Streams	Event Hubs	Services that facilitate the mass ingestion of events (messages), typically from devices and sensors. The data can then be processed in real-time micro-batches or be written to storage for further analysis.
IoT Things Graph	Digital Twins	Services you can use to create digital representations of real-world things, places, business processes, and people. Use these services to gain insights, drive the creation of better products and new customer experiences, and optimize operations and costs.

IOT architectures

-

IoT Architecture ◆ Azure IoT Subsystems

- 12/16/2019
- 1 min read

Learn about our recommended IoT application architecture that supports hybrid cloud and edge computing. A flowchart details how the subsystems function within the IoT application.

- Azure IoT reference architecture

  - 9/10/2020
  - 12 min read

  Recommended architecture for IoT applications on Azure using PaaS (platform-as-a-service) components

- Process real-time vehicle data using IoT

  - 11/17/2020
  - 5 min read

  This example builds a real-time data ingestion/processing pipeline to ingest and process messages from IoT devices into a big data analytic platform in Azure.

view all

Management and governance

AWS SERVICE	AZURE SERVICE	DESCRIPTION
AWS Organizations	Management Groups	Azure management groups help you organize your resources and subscriptions.
AWS Well-Architected Tool	Azure Well-Architected Review	Examine your workload through the lenses of reliability, cost management, operational excellence, security, and performance efficiency.
Trusted Advisor	Advisor	Provides analysis of cloud resource configuration and security, so that subscribers can ensure they're making use of best practices and optimum configurations.
AWS Billing and Cost Management	Azure Cost Management and Billing	Azure Cost Management and Billing helps you understand your Azure invoice (bill), manage your billing account and subscriptions, monitor and control Azure spending, and optimize resource use.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Cost and Usage Reports	Usage Details API	Services to help generate, monitor, forecast, and share billing data for resource usage by time, organization, or product resources.
Management Console	Portal	A unified management console that simplifies building, deploying, and operating your cloud resources.
Application Discovery Service	Migrate	Assesses on-premises workloads for migration to Azure, performs performance-based sizing, and provides cost estimations.
Systems Manager	Monitor	Comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments.
Personal Health Dashboard	Resource Health	Provides detailed information about the health of resources, as well as recommended actions for maintaining resource health.
CloudTrail	Activity log	The Activity log is a platform log in Azure that provides insight into subscription-level events, such as when a resource is modified or when a virtual machine is started.
CloudWatch	Application Insights	A feature of Azure Monitor, Application Insights is an extensible Application Performance Management (APM) service for developers and DevOps professionals, which provides telemetry insights and information, in order to better understand how applications are performing and to identify areas for optimization.
Config	Application Change Analysis	Application Change Analysis detects various types of changes, from the infrastructure layer all the way to application deployment.
Cost Explorer	Cost Management	Optimize costs while maximizing cloud potential.
Control Tower	Blueprints	Set up and govern a multi account/subscription environment by creating landing zones.
Resource Groups and Tag Editor	Resource Groups and Tags	A Resource Group is a container that holds related resources for an Azure solution. Apply tags to your Azure resources to logically organize them by categories.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
AWS AppConfig	Azure App Configuration	Azure App Configuration is a managed service that helps developers centralize their application and feature settings simply and securely.
Service Catalog	Azure Managed Applications	Offers cloud solutions that are easy for consumers to deploy and operate.
SDKs and tools	SDKs and tools	Manage and interact with Azure services the way you prefer, programmatically from your language of choice, by using the Azure SDKs, our collection of tools, or both.

Messaging and eventing

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Simple Queue Service (SQS)	Queue Storage	Provides a managed message queueing service for communicating between decoupled application components.
Simple Notification Service (SNS)	Service Bus	Supports a set of cloud-based, message-oriented middleware technologies, including reliable message queuing and durable publish/subscribe messaging.
Amazon EventBridge	Event Grid	A fully managed event routing service that allows for uniform event consumption using a publish/subscribe model.
Amazon Kinesis	Event Hub	A fully managed, real-time data ingestion service. Stream millions of events per second, from any source, to build dynamic data pipelines and to immediately respond to business challenges.
Amazon MQ	Service Bus	Service Bus Premium is fully compliant with the Java/Jakarta EE Java Message Service (JMS) 2.0 API. Service Bus Standard supports the JMS 1.1 subset focused on queues.

Messaging architectures

-

Anomaly Detector Process

- 12/16/2019
- 1 min read

Learn more about Anomaly Detector with a step-by-step flowchart that details the process. See how anomaly detection models are selected with time-series data.

-

Scalable web application

- 10/03/2019
- 7 min read

Use the proven practices in this reference architecture to improve scalability and performance in an Azure App Service web application..

-

Enterprise integration using queues and events

- 12/03/2018
- 5 min read

Recommended architecture for implementing an enterprise integration pattern with Azure Logic Apps, Azure API Management, Azure Service Bus, and Azure Event Grid.

-

Ops automation using Event Grid

- 12/16/2019
- 1 min read

Event Grid allows you to speed automation and simplify policy enforcement. For example, Event Grid can notify Azure Automation when a virtual machine is created, or a SQL Database is spun up. These events can be used to automatically check that service configurations are compliant, put metadata into operations tools, tag virtual machines, or file work items.

Mobile services

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Mobile Hub	App Center Xamarin Apps	Provides backend mobile services for rapid development of mobile solutions, identity management, data synchronization, and storage and notifications across devices.
Mobile SDK	App Center	Provides the technology to rapidly build cross-platform and native apps for mobile devices.
Cognito	Azure Active Directory	Provides authentication capabilities for mobile applications.
Device Farm	App Center	Provides services to support testing mobile applications.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Mobile Analytics	App Center	Supports monitoring, and feedback collection for the debugging and analysis of a mobile application service quality.

Device Farm

The AWS Device Farm provides cross-device testing services. In Azure, Visual Studio App Center provides similar cross-device front-end testing for mobile devices.

In addition to front-end testing, the Azure DevTest Labs provides back-end testing resources for Linux and Windows environments.

Mobile architectures

- Scalable web and mobile applications using Azure Database for PostgreSQL

  - 12/16/2019
  - 1 min read

  Use Azure Database for PostgreSQL to rapidly build engaging, performant, and scalable cross-platform and native apps for iOS, Android, Windows, or Mac.

- Social App for Mobile and Web with Authentication

  - 12/16/2019
  - 3 min read

  View a detailed, step-by-step diagram depicting the build process and implementation of the mobile client app architecture that offers social image sharing with a companion web app and authentication abilities, even while offline.

- Task-Based Consumer Mobile App

  - 12/16/2019
  - 3 min read

  Learn how the task-based consumer mobile app architecture is created with a step-by-step flow chart that shows the integration with Azure App Service Mobile Apps, Visual Studio, and Xamarin to simplify the build process.

view all

Networking

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Cloud virtual networking	Virtual Private Cloud (VPC)	Virtual Network	Provides an isolated, private environment in the cloud. Users have control over their virtual networking environment, including selection of their own IP address range, creation of subnets, and configuration of route tables and network gateways.
NAT gateways	NAT Gateways	Virtual Network NAT	A service that simplifies outbound-only Internet connectivity for virtual networks. When configured on a subnet, all outbound connectivity uses your specified static public IP addresses. Outbound connectivity is possible without a load balancer or public IP addresses directly attached to virtual machines.
Cross-premises connectivity	VPN Gateway	VPN Gateway	Connects Azure virtual networks to other Azure virtual networks, or customer on-premises networks (Site To Site). Allows end users to connect to Azure services through VPN tunneling (Point To Site).
DNS management	Route 53	DNS	Manage your DNS records using the same credentials and billing and support contract as your other Azure services
DNS-based routing	Route 53	Traffic Manager	A service that hosts domain names, plus routes users to Internet applications, connects user requests to datacenters, manages traffic to apps, and improves app availability with automatic failover.
Dedicated network	Direct Connect	ExpressRoute	Establishes a dedicated, private network connection from a location to the cloud provider (not over the Internet).

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Load balancing	Network Load Balancer	Load Balancer	Azure Load Balancer load balances traffic at layer 4 (TCP or UDP). Standard Load Balancer also supports cross-region or global load balancing.
Application-level load balancing	Application Load Balancer	Application Gateway	Application Gateway is a layer 7 load balancer. It supports SSL termination, cookie-based session affinity, and round robin for load-balancing traffic.
Route table	Custom Route Tables	User Defined Routes	Custom, or user-defined (static) routes to override default system routes, or to add more routes to a subnet's route table.
Private link	PrivateLink	Azure Private Link	Azure Private Link provides private access to services that are hosted on the Azure platform. This keeps your data on the Microsoft network.
Private PaaS connectivity	VPC endpoints	Private Endpoint	Private Endpoint provides secured, private connectivity to various Azure platform as a service (PaaS) resources, over a backbone Microsoft private network.
Virtual network peering	VPC Peering	VNET Peering	VNet peering is a mechanism that connects two virtual networks (VNets) in the same region through the Azure backbone network. Once peered, the two virtual networks appear as one for all connectivity purposes.
Content delivery networks	Cloud Front	Azure CDN	The Azure Content Delivery Network is designed to send audio, video, apps, photos, and other files to your customers faster and more reliably, using the servers closest to each user. Acceleration Data Transfer provides dynamic site acceleration of non-cacheable, dynamic content that is generated by your web applications.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Network Monitoring	VPC Flow Logs	Azure Network Watcher	Azure Network Watcher allows you to monitor, diagnose, and analyze the traffic in Azure Virtual Network.

Networking architectures

- Deploy highly available NVAs
  - 12/08/2018
  - 7 min read

  Learn how to deploy network virtual appliances for high availability in Azure. This article includes example architectures for ingress, egress, and both.
- Hub-spoke network topology in Azure
  - 9/30/2020
  - 7 min read

  Learn how to implement a hub-spoke topology in Azure, where the hub is a virtual network and the spokes are virtual networks that peer with the hub.
- Implement a secure hybrid network
  - 1/07/2020
  - 9 min read

  See a secure hybrid network that extends an on-premises network to Azure with a perimeter network between the on-premises network and an Azure virtual network.

view all

Security, identity, and access

Authentication and authorization

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Identity and Access Management (IAM)	Azure Active Directory	Allows users to securely control access to services and resources while offering data security and protection. Create and manage users and groups, and use permissions to allow and deny access to resources.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Identity and Access Management (IAM)	Azure role-based access control	Azure role-based access control (Azure RBAC) helps you manage who has access to Azure resources, what they can do with those resources, and what areas they have access to.
Organizations	Subscription Management + Azure RBAC	Security policy and role management for working with multiple accounts.
Multi-Factor Authentication	Azure Active Directory	Safeguard access to data and applications, while meeting user demand for a simple sign-in process.
Directory Service	Azure Active Directory Domain Services	Provides managed domain services, such as domain join, group policy, LDAP, and Kerberos/NTLM authentication, which are fully compatible with Windows Server Active Directory.
Cognito	Azure Active Directory B2C	A highly available, global, identity management service for consumer-facing applications that scales to hundreds of millions of identities.
Organizations	Policy	Azure Policy is a service in Azure that you use to create, assign, and manage policies. These policies enforce different rules and effects over your resources, so those resources stay compliant with your corporate standards and service level agreements.
Organizations	Management Groups	Azure management groups provide a level of scope above subscriptions. You organize subscriptions into containers called "management groups" and apply your governance conditions to the management groups. All subscriptions within a management group automatically inherit the conditions applied to the management group. Management groups give you enterprise-grade management at a large scale, no matter what type of subscriptions you have.

Encryption

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Server-side encryption with Amazon S3 Key Management Service	Azure Storage Service Encryption	Helps you protect and safeguard your data and meet your organizational security and compliance commitments.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Key Management Service (KMS), CloudHSM	Key Vault	Provides security solution and works with other services by providing a way to manage, create, and control encryption keys stored in hardware security modules (HSM).

Firewall

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Web Application Firewall	Web Application Firewall	A firewall that protects web applications from common web exploits.
Web Application Firewall	Firewall	Provides inbound protection for non-HTTP/S protocols, outbound network-level protection for all ports and protocols, and application-level protection for outbound HTTP/S.

Security

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Inspector	Defender for Cloud	An automated security assessment service that improves the security and compliance of applications. Automatically assess applications for vulnerabilities or deviations from best practices.
Certificate Manager	App Service Certificates available on the Portal	Service that allows customers to create, manage, and consume certificates seamlessly in the cloud.
GuardDuty	Advanced Threat Protection	Detect and investigate advanced attacks on-premises and in the cloud.
Artifact	Service Trust Portal	Provides access to audit reports, compliance guides, and trust documents from across cloud services.
Shield	DDos Protection Service	Provides cloud services with protection from distributed denial of services (DDoS) attacks.

Security architectures

-

Real-time fraud detection

- 7/05/2018
- 4 min read

Detect fraudulent activity in real-time using Azure Event Hubs and Stream Analytics.

-

  Securely managed web applications

  - 5/09/2019
  - 8 min read

  Learn about deploying secure applications using the Azure App Service Environment, the Azure Application Gateway service, and Web Application Firewall.

-

  Threat indicators for cyber threat intelligence in Azure Sentinel

  - 4/13/2020
  - 13 min read

  Import threat indicators, view logs, create rules to generate security alerts and incidents, and visualize threat intelligence data with Azure Sentinel.

view all

Storage

Object storage

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Simple Storage Services (S3)	Blob storage	Object storage service, for use cases including cloud applications, content distribution, backup, archiving, disaster recovery, and big data analytics.

Virtual server disks

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Block Store (EBS)	managed disks	SSD storage optimized for I/O intensive read/write operations. For use as high-performance Azure virtual machine storage.

Shared files

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic File System	Files	Provides a simple interface to create and configure file systems quickly, and share common files. Can be used with traditional protocols that access files over a network.

Archiving and backup

AWS SERVICE	AZURE SERVICE	DESCRIPTION
S3 Infrequent Access (IA)	Storage cool tier	Cool storage is a lower-cost tier for storing data that is infrequently accessed and long-lived.
S3 Glacier, Deep Archive	Storage archive access tier	Archive storage has the lowest storage cost and higher data retrieval costs compared to hot and cool storage.
Backup	Backup	Back up and recover files and folders from the cloud, and provide offsite protection against data loss.

Hybrid storage

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Storage Gateway	StorSimple	Integrates on-premises IT environments with cloud storage. Automates data management and storage, plus supports disaster recovery.
DataSync	File Sync	Azure Files can be deployed in two main ways: by directly mounting the serverless Azure file shares or by caching Azure file shares on-premises using Azure File Sync.

Bulk data transfer

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Import/Export Disk	Import/Export	A data transport solution that uses secure disks and appliances to transfer large amounts of data. Also offers data protection during transit.
Import/Export Snowball, Snowball Edge, Snowmobile	Data Box	Petabyte- to exabyte-scale data transport solution that uses secure data storage devices to transfer large amounts of data to and from Azure.

Storage architectures

-

  HIPAA and HITRUST compliant health data AI

  - 12/16/2019
  - 2 min read

  Manage HIPAA and HITRUST compliant health data and medical records with the highest level of built-in security.

-

HPC Media Rendering

- 11/04/2020
- 2 min read

Optimize the media rendering process with a step-by-step HPC solution architecture from Azure that combines Azure CycleCloud and HPC Cache.

-

Medical Data Storage Solutions

- 12/16/2019
- 2 min read

Store healthcare data effectively and affordably with cloud-based solutions from Azure. Manage medical records with the highest level of built-in security.

view all

Web applications

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Elastic Beanstalk	App Service	Managed hosting platform providing easy to use services for deploying and scaling web applications and services.
API Gateway	API Management	A turnkey solution for publishing APIs to external and internal consumers.
CloudFront	Content Delivery Network	A global content delivery network that delivers audio, video, applications, images, and other files.
Global Accelerator	Front Door	Easily join your distributed microservices architectures into a single global application using HTTP load balancing and path-based routing rules. Automate turning up new regions and scale-out with API-driven global actions, and independent fault-tolerance to your back end microservices in Azure-or anywhere.
Global Accelerator	Cross-regional load balancer	Distribute and load balance traffic across multiple Azure regions via a single, static, global anycast public IP address.
LightSail	App Service	Build, deploy, and scale web apps on a fully managed platform.
App Runner	Web App for Containers	Easily deploy and run containerized web apps on Windows and Linux.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
Amplify	Static Web Apps	Boost productivity with a tailored developer experience, CI/CD workflows to build and deploy your static content hosting, and dynamic scale for integrated serverless APIs.

Web architectures

- Architect scalable e-commerce web app
  - 12/16/2019
  - 1 min read

  The e-commerce website includes simple order processing workflows with the help of Azure services. Using Azure Functions and Web Apps, developers can focus on building personalized experiences and let Azure take care of the infrastructure.
- Multi-region N-tier application
  - 6/18/2019
  - 10 min read

  Deploy an application on Azure virtual machines in multiple regions for high availability and resiliency.
- Serverless web application
  - 5/28/2019
  - 16 min read

  This reference architecture shows a serverless web application, which serves static content from Azure Blob Storage and implements an API using Azure Functions.

view all

End-user computing

AWS SERVICE	AZURE SERVICE	DESCRIPTION
WorkSpaces, AppStream 2.0	Azure Virtual Desktop	Manage virtual desktops and applications to enable corporate network and data access to users, anytime, anywhere, from supported devices. Amazon WorkSpaces support Windows and Linux virtual desktops. Azure Virtual Desktop supports multi-session Windows 10 virtual desktops.

AWS SERVICE	AZURE SERVICE	DESCRIPTION
WorkLink	Application Proxy	Provides access to intranet applications, without requiring VPN connectivity. Amazon WorkLink is limited to iOS and Android devices.

Miscellaneous

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Backend process logic	Step Functions	Logic Apps	Cloud technology to build distributed applications using out-of-the-box connectors to reduce integration challenges. Connect apps, data, and devices on-premises or in the cloud.
Enterprise application services	WorkMail, WorkDocs, Chime	Microsoft 365	Fully integrated cloud service that provides communications, email, and document management in the cloud and is available on a wide variety of devices.
Gaming	GameLift	PlayFab	Managed services for hosting dedicated game servers.
Media transcoding	Elastic Transcoder	Media Services	Services that offer broadcast-quality video streaming services, including various transcoding technologies.
Workflow	Simple Workflow Service (SWF)	Logic Apps	Serverless technology for connecting apps, data and devices anywhere, whether on-premises or in the cloud for large ecosystems of SaaS and cloud-based connectors.
Hybrid	Outposts	Stack	Azure Stack is a hybrid cloud platform that enables you to run Azure services in your company's or service provider's datacenter. As a developer, you can build apps on Azure Stack. You can then deploy them to either Azure Stack or Azure, or you can build truly hybrid apps that take advantage of connectivity between an Azure Stack cloud and Azure.

AREA	AWS SERVICE	AZURE SERVICE	DESCRIPTION
Media	Elemental MediaConvert	Media Services	Cloud-based media workflow platform to index, package, protect, and stream video at scale.
Satellite	Ground Station	Azure Orbital	Fully managed cloud-based ground station as a service.
Quantum computing	Amazon Braket	Azure Quantum	Managed quantum computing service that developers, researchers, and businesses can use to run quantum computing programs.

More learning

If you are new to Azure, review the interactive Core Cloud Services - Introduction to Azure module on Microsoft Learn.

Azure for Google Cloud Professionals

11/19/2021 • 9 minutes to read • Edit Online

This article helps Google Cloud experts understand the basics of Microsoft Azure accounts, platform, and services. It also covers key similarities and differences between the Google Cloud and Azure platforms. (Note that Google Cloud was previously called Google Cloud Platform (GCP).)

You'll learn:

- How accounts and resources are organized in Azure.
- How available solutions are structured in Azure.
- How the major Azure services differ from Google Cloud services.

Azure and Google Cloud built their capabilities independently over time so that each has important implementation and design differences.

Azure for Google Cloud overview

Like Google Cloud, Microsoft Azure is built around a core set of compute, storage, database, and networking services. In many cases, both platforms offer a basic equivalence between the products and services they offer. Both Google Cloud and Azure allow you to build highly available solutions based on Linux or Windows hosts. So, if you're used to development using Linux and OSS technology, both platforms can do the job.

While the capabilities of both platforms are similar, the resources that provide those capabilities are often organized differently. Exact one-to-one relationships between the services required to build a solution are not always clear. In other cases, a particular service might be offered on one platform, but not the other.

Managing accounts and subscription

Azure has a hierarchy of Management group and subscriptions and resource groups to manage resources effectively. This is similar to the Folders and Project hierarchy for resources in Google Cloud.

Azure levels of management scope

- **Management groups:** These groups are containers that help you manage access, policy, and compliance for multiple subscriptions. All subscriptions in a management group automatically inherit the conditions applied to the management group.
- **Subscriptions:** A subscription logically associates user accounts and the resources that were created by

those user accounts. Each subscription has limits or quotas on the amount of resources you can create and use. Organizations can use subscriptions to manage costs and the resources that are created by users, teams, or projects.

- **Resource groups:** A resource group is a logical container into which Azure resources like web apps, databases, and storage accounts are deployed and managed.
- **Resources:** Resources are instances of services that you create, like virtual machines, storage, or SQL databases.

Azure services can be purchased using several pricing options, depending on your organization's size and needs. See the pricing overview page for details.

Azure subscriptions are a grouping of resources with an assigned owner responsible for billing and permissions management.

A GCP *project* is conceptually similar to the Azure subscription, in terms of billing, quotas, and limits. However, from a functional perspective, a GCP project is more like a resource group in Azure. It's a logical unit that cloud resources are deployed to.

Note that unlike in GCP, there is no maximum number of Azure subscriptions. Each Azure subscription is linked to a single Azure Active Directory (Azure AD) tenant (an *account*, in GCP terms). An Azure AD tenant can contain an unlimited number of subscriptions, whereas GCP has a default limit of 30 projects per account.

Subscriptions are assigned three types of administrator accounts:

- **Account Administrator**. The subscription owner and the account billed for the resources used in the subscription. The account administrator can only be changed by transferring ownership of the subscription.
- **Service Administrator**. This account has rights to create and manage resources in the subscription but is not responsible for billing. By default, the account administrator and service administrator are assigned to the same account. The account administrator can assign a separate user to the service administrator account for managing the technical and operational aspects of a subscription. Only one service administrator is assigned per subscription.
- **Co-administrator**. There can be multiple co-administrator accounts assigned to a subscription. Co-administrators cannot change the service administrator, but otherwise have full control over subscription resources and users.

For fine-grained access management to Azure resources, you can use Azure role-based access control (Azure RBAC), which includes over 70 built-in roles. You can also create your own custom roles.

Below the subscription level user roles and individual permissions can also be assigned to specific resources. In Azure, all user accounts are associated with either a Microsoft Account or Organizational Account (an account managed through Azure AD).

Subscriptions have default service quotas and limits. For a full list of these limits, see Azure subscription and service limits, quotas, and constraints. These limits can be increased up to the maximum by filing a support request in the management portal.

See also

- How to add or change Azure administrator roles
- How to download your Azure billing invoice and daily usage data

Resource management

The term "resource" in Azure means any compute instance, storage object, networking device, or other entity you can create or configure within the platform.

Azure resources are deployed and managed using one of two models: Azure Resource Manager, or the older

Azure classic deployment model. Any new resources are created using the Resource Manager model.

Resource groups

Azure additionally has an entity called "resource groups" that organize resources such as VMs, storage, and virtual networking devices. An Azure resource is always associated with one resource group. A resource created in one resource group can be moved to another group but can only be in one resource group at a time. For more information, see Move Azure resources across resource groups, subscriptions, or regions. Resource groups are the fundamental grouping used by Azure Resource Manager.

Resources can also be organized using tags. Tags are key-value pairs that allow you to group resources across your subscription irrespective of resource group membership.

Management interfaces

Azure offers several ways to manage your resources:

- Web interface. The Azure portal provides a full web-based management interface for Azure resources.
- REST API. The Azure Resource Manager REST API provides programmatic access to most of the features available in the Azure portal.
- Command Line. The Azure CLI provides a command-line interface capable of creating and managing Azure resources. The Azure CLI is available for Windows, Linux, and macOS.
- PowerShell. The Azure modules for PowerShell allow you to execute automated management tasks using a script. PowerShell is available for Windows, Linux, and macOS.
- Templates. Azure Resource Manager templates provide JSON template-based resource management capabilities.
- SDK. The SDKs are a collection of libraries that allows users to programmatically manage and interact with Azure services.

In each of these interfaces, the resource group is central to how Azure resources get created, deployed, or modified.

In addition, many third-party management tools like Hashicorp's Terraform and Netflix Spinnaker, are also available on Azure.

See also

- Azure resource group guidelines

Regions and Availability Zones

Failures can vary in the scope of their impact. Some hardware failures, such as a failed disk, may affect a single host machine. A failed network switch could affect a whole server rack. Less common are failures that disrupt a whole datacenter, such as loss of power in a datacenter. In rare situations, an entire region could become unavailable.

One of the main ways to make an application resilient is through redundancy. However, you need to plan for this redundancy when you design the application. Also, the level of redundancy that you need depends on your business requirements. Not every application needs redundancy across regions to guard against a regional outage. In general, a tradeoff exists between greater redundancy and reliability versus higher cost and complexity.

In Google Cloud, a region has two or more Availability Zones. An Availability Zone corresponds with a physically isolated datacenter in the geographic region. Azure has numerous features for providing application redundancy at every level of potential failure, including **availability sets** , **availability zones** , and **paired regions**.

The following table summarizes each option.

	AVAILABILITY SET	AVAILABILITY ZONE	PAIRED REGION
Scope of failure	Rack	Datacenter	Region
Request routing	Load Balancer	Cross-zone Load Balancer	Traffic Manager
Network latency	Very low	Low	Mid to high
Virtual networking	VNet	VNet	Cross-region VNet peering

Availability sets

To protect against localized hardware failures, such as a disk or network switch failing, deploy two or more VMs in an availability set. An availability set consists of two or more *fault domains* that share a common power source and network switch. VMs in an availability set are distributed across the fault domains, so if a hardware failure affects one fault domain, network traffic can still be routed the VMs in the other fault domains. For more information about Availability Sets, see Manage the availability of Windows virtual machines in Azure.

When VM instances are added to availability sets, they are also assigned an update domain. An update domain is a group of VMs that are set for planned maintenance events at the same time. Distributing VMs across multiple update domains ensures that planned update and patching events affect only a subset of these VMs at any given time.

Availability sets should be organized by the instance's role in your application to ensure one instance in each role is operational. For example, in a three-tier web application, create separate availability sets for the front-end, application, and data tiers.

Availability sets

Availability Zones

Like Google Cloud, Azure regions can have Availability zones. An Availability Zone is a physically separate zone within an Azure region. Each Availability Zone has a distinct power source, network, and cooling. Deploying VMs across availability zones helps to protect an application against datacenter-wide failures.

Zone redundant VM deployment on Azure

For more information, see Build solutions for high availability using Availability Zones.

Paired regions

To protect an application against a regional outage, you can deploy the application across multiple regions, using Azure Traffic Manager to distribute internet traffic to the different regions. Each Azure region is paired with another region. Together, these form a regional pair. With the exception of Brazil South, regional pairs are located within the same geography in order to meet data residency requirements for tax and law enforcement jurisdiction purposes.

Unlike Availability Zones, which are physically separate datacenters but may be in relatively nearby geographic areas, paired regions are typically separated by at least 300 miles. This design ensures that large-scale disasters only affect one of the regions in the pair. Neighboring pairs can be set to sync database and storage service data, and are configured so that platform updates are rolled out to only one region in the pair at a time.

Azure geo-redundant storage is automatically backed up to the appropriate paired region. For all other resources, creating a fully redundant solution using paired regions means creating a full copy of your solution in both regions.

Region Pairs in Azure

See also

- Regions for virtual machines in Azure
- Availability options for virtual machines in Azure
- High availability for Azure applications
- Failure and disaster recovery for Azure applications

- Planned maintenance for Linux virtual machines in Azure

Services

For a listing of how services map between platforms, see Google Cloud to Azure services comparison.

Not all Azure products and services are available in all regions. Consult the Products by Region page for details. You can find the uptime guarantees and downtime credit policies for each Azure product or service on the Service Level Agreements page.

Next steps

- Get started with Azure
- Azure solution architectures
- Azure Reference Architectures

Google Cloud to Azure services comparison

11/19/2021 • 31 minutes to read • Edit Online

This article helps you understand how Microsoft Azure services compare to Google Cloud. (Note that Google Cloud used to be called the Google Cloud Platform (GCP).) Whether you are planning a multi-cloud solution with Azure and Google Cloud, or migrating to Azure, you can compare the IT capabilities of Azure and Google Cloud services in all the technology categories.

This article compares services that are roughly comparable. Not every Google Cloud service or Azure service is listed, and not every matched service has exact feature-for-feature parity.

For an overview of Azure for Google Cloud users, see the introduction to Azure for Google Cloud Professionals.

Marketplace

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Google Cloud Marketplace	Azure Marketplace	Easy-to-deploy and automatically configured third-party applications, including single virtual machine or multiple virtual machine solutions.

Data platform

Database

TYPE	GOOGLE CLOUD SERVICE	AZURE SERVICE	AZURE SERVICE DESCRIPTION
Relational database	Cloud SQL - SQL Server	Azure SQL Server Family Azure SQL Database Azure SQL Managed Instance SQL Server on Azure VM Azure SQL Edge	Azure SQL family of SQL Server database engine products in the cloud Azure SQL Database is a fully managed platform as a service (PaaS) database engine Azure SQL Managed Instance is the intelligent, scalable cloud database service that combines the broadest SQL Server database engine compatibility with all the benefits of a fully managed and evergreen platform as a service SQL Server IaaS deployed on Azure Windows or Linux VM Azure SQL Edge is an optimized relational database engine geared for IoT and edge deployments

TYPE	GOOGLE CLOUD SERVICE	AZURE SERVICE	AZURE SERVICE DESCRIPTION
	Cloud SQL MySQL & PostgreSQL	Azure Database for MySQL (Single & Flexible Server) Azure Database for PostgreSQL (Single & Flexible Server)	Managed relational database service where resiliency, security, scale, and maintenance are primarily handled by the platform
Horizontally scalable relational database	Cloud Spanner	Azure Cosmos DB - SQL API	A globally-distributed database system that limitlessly scales horizontally. Is multi-modal -- key-value, graph, and document data). Supports multiple APIs: SQL, JavaScript, Gremlin, MongoDB, and Azure Table storage. Compute and storage can be scaled independently
		Azure PostgreSQL Hyperscale (Citus)	Azure Database for PostgreSQL is a fully managed database-as-a-service based on the open-source Postgres relational database engine. The Hyperscale (Citus) deployment option scales queries across multiple machines using sharding, to serve applications that require greater scale and performance
NoSQL	Cloud Bigtable	Azure Table storage	A highly scalable NoSQL key-value store for rapid development using massive semi-structured datasets. Store semi-structured data that's highly available. Supporting flexible data schema and OData-based queries
	Cloud Firestore	Azure Cosmos DB	Globally distributed, multi-model database that natively supports multiple data models: key-value, documents, graphs, and columnar

TYPE	GOOGLE CLOUD SERVICE	AZURE SERVICE	AZURE SERVICE DESCRIPTION
	Firebase Realtime Database	Azure Cosmos DB - Change Feed	Change feed in Azure Cosmos DB is a persistent record of changes to a container in the order they occur. Change feed support in Azure Cosmos DB works by listening to an Azure Cosmos container for any changes. It then outputs the sorted list of documents that were changed in the order in which they were modified. The persisted changes can be processed asynchronously and incrementally, and the output can be distributed across one or more consumers for parallel processing
In-memory	Cloud Memorystore	Azure Cache for Redis	A secure data cache and messaging broker that provides high throughput and low-latency access to data for applications

Database architectures

-

  Gaming using Cosmos DB

  - 12/16/2019
  - 1 min read

  Elastically scale your database to accommodate unpredictable bursts of traffic and deliver low-latency multi-player experiences on a global scale.

-

  Oracle Database Migration to Azure

  - 12/16/2019
  - 2 min read

  Oracle DB migrations can be accomplished in multiple ways. This architecture covers one of these options wherein Oracle Active Data Guard is used to migrate the Database.

-

  Retail and e-commerce using Azure MySQL

  - 12/16/2019

- 1 min read

Build secure and scalable e-commerce solutions that meet the demands of both customers and business using Azure Database for MySQL.

view all

Data warehouse

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
BigQuery	Azure Synapse Analytics SQL Server Big Data Clusters Azure Databricks	Cloud-based Enterprise Data Warehouse (EDW) that uses Massively Parallel Processing (MPP) to quickly run complex queries across petabytes of data. Allow you to deploy scalable clusters of SQL Server, Spark, and HDFS containers running on Kubernetes. These components are running side by side to enable you to read, write, and process big data from Transact-SQL or Spark, allowing you to easily combine and analyze your high-value relational data with high-volume big data.

Data warehouse architectures

- Modern Data Warehouse Architecture

  - 12/16/2019
  - 2 min read

  Explore a cloud data warehouse that uses big data. Modern data warehouse brings together all your data and scales easily as your data grows.

- Automated enterprise BI

  - 6/03/2020
  - 13 min read

  Automate an extract, load, and transform (ELT) workflow in Azure using Azure Data Factory with Azure Synapse Analytics.

view all

Data orchestration and ETL

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Data Fusion	Azure Data Factory Azure Synapse Analytics	Processes and moves data between different compute and storage services, as well as on-premises data sources at specified intervals. Create, schedule, orchestrate, and manage data pipelines.

Big data and analytics

Big data processing

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Dataproc	Azure HDInsight Azure Synapse Analytics Azure Databricks	Managed Apache Spark-based analytics platform.

Big data architectures

- Azure data platform end-to-end
  - 1/31/2020
  - 7 min read

  Use Azure services to ingest, process, store, serve, and visualize data from different sources.
- Campaign Optimization with Azure HDInsight Spark Clusters
  - 12/16/2019
  - 4 min read

  This solution demonstrates how to build and deploy a machine learning model with Microsoft R Server on Azure HDInsight Spark clusters to recommend actions to maximize the purchase rate of leads targeted by a campaign. This solution enables efficient handling of big data on Spark with Microsoft R Server.
- Big data analytics with Azure Data Explorer
  - 8/11/2020
  - 3 min read

  Big Data Analytics with Azure Data Explorer demonstrates Azure Data Explorer's abilities to cater to volume, velocity, and variety of data, the three V's of big data.

view all

Analytics and visualization

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Dataflow	Azure Databricks	Managed platform for streaming batch data based on Open Source Apache products.
Data Studio Looker	Power BI	Business intelligence tools that build visualizations, perform ad hoc analysis, and develop business insights from data.
Cloud Search	Azure Search	Delivers full-text search and related search analytics and capabilities.
BigQuery	SQL Server Analysis Services	Provides a serverless non-cloud interactive query service that uses standard SQL for analyzing databases.

Analytics architectures

- Advanced Analytics Architecture

  - 12/16/2019
  - 2 min read

  Get near real-time data analytics on streaming services. This big data architecture allows you to combine any data at any scale with custom machine learning.

- Automated enterprise BI

  - 6/03/2020
  - 13 min read

  Automate an extract, load, and transform (ELT) workflow in Azure using Azure Data Factory with Azure Synapse Analytics.

- Mass ingestion and analysis of news feeds on Azure

  - 2/01/2019
  - 5 min read

  Create a pipeline for ingesting and analyzing text, images, sentiment, and other data from RSS news feeds using only Azure services, including Azure Cosmos DB and Azure Cognitive Services.

view all

Time series & IOT data

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
BigQuery	Azure Data Explorer Azure Time Series Insights Cosmos DB	Fully managed, low latency, and distributed big data analytics platform that runs complex queries across petabytes of data. Highly optimized for log and time series data. Open and scalable end-to-end IoT analytics service. Collect, process, store, query, and visualize data at Internet of Things (IoT) scale--data that's highly contextualized and optimized for time series.

Time series architectures

- IoT analytics with Azure Data Explorer

  - 8/11/2020
  - 3 min read

  IoT Telemetry Analytics with Azure Data Explorer demonstrates near real-time analytics over fast flowing, high volume, wide variety of streaming data from IoT devices.

- Azure Data Explorer interactive analytics

  - 8/11/2020
  - 3 min read

  Interactive Analytics with Azure Data Explorer focuses on its integration with the rest of the data platform ecosystem.

AI and machine learning

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Vertex AI	Azure Machine Learning	A cloud service to train, deploy, automate, and manage machine learning models.
TensorFlow	ML.NET	ML.NET is an open source and cross-platform machine learning framework for both machine learning & AI.
TensorFlow	ONNX (Open Neural Network Exchange)	ONNX is an open format built to represent machine learning models that facilitates maximum compatibility and increased inference performance.

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Vision AI	Azure Cognitive Services Computer Vision	Use visual data processing to label content, from objects to concepts, extract printed and handwritten text, recognize familiar subjects like brands and landmarks, and moderate content. No machine learning expertise is required.
Natural Language AI	Azure Cognitive Services Text Analytics	Cloud-based services that provides advanced natural language processing over raw text, and includes four main functions: sentiment analysis, key phrase extraction, language detection, and named entity recognition.
Natural Language AI	Azure Cognitive Services Language Understanding (LUIS)	A machine learning-based service to build natural language understanding into apps, bots, and IoT devices. Quickly create enterprise-ready, custom models that continuously improve.
Speech-to-Text	Azure Cognitive Services Speech To Text	Swiftly convert audio into text from a variety of sources. Customize models to overcome common speech recognition barriers, such as unique vocabularies, speaking styles, or background noise.
AutoML Tables – Structured Data	Azure ML - Automated Machine Learning	Empower professional and non-professional data scientists to build machine learning models rapidly. Automate time-consuming and iterative tasks of model development using breakthrough research-and accelerate time to market. Available in Azure Machine learning, Power BI, ML.NET & Visual Studio.
AutoML Tables – Structured Data	ML.NET Model Builder	ML.NET Model Builder provides an easy to understand visual interface to build, train, and deploy custom machine learning models. Prior machine learning expertise is not required. Model Builder supports AutoML, which automatically explores different machine learning algorithms and settings to help you find the one that best suits your scenario.
AutoML Vision	Azure Cognitive Services Custom Vision	Customize and embed state-of-the-art computer vision for specific domains. Build frictionless customer experiences, optimize manufacturing processes, accelerate digital marketing campaigns-and more. No machine learning expertise is required.

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
AutoML Video Intelligence	Azure Video Analyzer	Easily extract insights from your videos and quickly enrich your applications to enhance discovery and engagement.
Dialogflow	Azure Cognitive Services QnA Maker	Build, train and publish a sophisticated bot using FAQ pages, support websites, product manuals, SharePoint documents or editorial content through an easy-to-use UI or via REST APIs.
AI Platform Notebooks	Azure Notebooks	Develop and run code from anywhere with Jupyter notebooks on Azure.
Deep Learning VM Image	Data Science Virtual Machines	Pre-Configured environments in the cloud for Data Science and AI Development.
Deep Learning Containers	GPU support on Azure Kubernetes Service (AKS)	Graphical processing units (GPUs) are often used for compute-intensive workloads such as graphics and visualization workloads. AKS supports the creation of GPU-enabled node pools to run these compute-intensive workloads in Kubernetes.
Data Labeling Service	Azure ML - Data Labeling	A central place to create, manage, and monitor labeling projects (public preview). Use it to coordinate data, labels, and team members to efficiently manage labeling tasks. Machine Learning supports image classification, either multi-label or multi-class, and object identification with bounded boxes.
AI Platform Training	Azure ML – Compute Targets	Designated compute resource/environment where you run your training script or host your service deployment. This location may be your local machine or a cloud-based compute resource. Using compute targets make it easy for you to later change your compute environment without having to change your code.
AI Platform Predictions	Azure ML - Deployments	Deploy your machine learning model as a web service in the Azure cloud or to Azure IoT Edge devices. Leverage serverless Azure Functions for model inference for dynamic scale.

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Continuous Evaluation	Azure ML – Data Drift	Monitor for data drift between the training dataset and inference data of a deployed model. In the context of machine learning, trained machine learning models may experience degraded prediction performance because of drift. With Azure Machine Learning, you can monitor data drift and the service can send an email alert to you when drift is detected.
What-If Tool	Azure ML – Model Interpretability	Ensure machine learning model compliance with company policies, industry standards, and government regulations.
Cloud TPU	Azure ML – FPGA (Field Programmable Gate Arrays)	FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects. The interconnects allow these blocks to be configured in various ways after manufacturing. Compared to other chips, FPGAs provide a combination of programmability and performance.
Kubeflow	Machine Learning Operations (MLOps)	MLOps, or DevOps for machine learning, enables data science and IT teams to collaborate and increase the pace of model development and deployment via monitoring, validation, and governance of machine learning models.
Dialogflow	Microsoft Bot Framework	Build and connect intelligent bots that interact with your users using text/SMS, Skype, Teams, Slack, Microsoft 365 mail, Twitter, and other popular services.

AI and machine learning architectures

- Image classification on Azure

  - 7/05/2018
  - 4 min read

  Learn how to build image processing into your applications by using Azure services such as the Computer Vision API and Azure Functions.

- Predictive Marketing with Machine Learning

  - 12/16/2019

- 2 min read

Learn how to build a machine-learning model with Microsoft R Server on Azure HDInsight Spark clusters to recommend actions to maximize the purchase rate.

-

Scalable personalization on Azure

- 5/31/2019
- 6 min read

Use machine learning to automate content-based personalization for customers.

view all

Data catalog & governance

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Data Catalog	Azure Purview	Azure Purview is a unified data governance service that helps you manage and govern your on-premises, multi-cloud, and software-as-a-service (SaaS) data.

Compute

Virtual servers

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Compute Engine	Azure Virtual Machines	Virtual servers allow users to deploy, manage, and maintain OS and server software. Instance types provide combinations of CPU/RAM. Users pay for what they use with the flexibility to change sizes.
Sole-tenant nodes	Azure Dedicated Host	Host your VMs on hardware that's dedicated only to your project.
Batch	Azure Batch	Run large-scale parallel and high-performance computing applications efficiently in the cloud.
Compute Engine Autoscaler Compute Engine managed instance groups	Azure virtual machine scale sets	Allows you to automatically change the number of VM instances. You set defined metric and thresholds that determine if the platform adds or removes instances.

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud GPUs	GPU Optimized VMs	GPU-optimized VM sizes are specialized virtual machines that are available with single, multiple, or fractional GPUs. The sizes are designed for compute-intensive, graphics-intensive, and visualization workloads.
VMware Engine	Azure VMware Solution	Redeploy and extend your VMware-based enterprise workloads to Azure with Azure VMware Solution. Seamlessly move VMware-based workloads from your datacenter to Azure and integrate your VMware environment with Azure. Keep managing your existing environments with the same VMware tools that you already know, while you modernize your applications with Azure native services. Azure VMware Solution is a Microsoft service that is verified by VMware, and it runs on Azure infrastructure.

Containers and container orchestrators

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Run	Azure Container Instances	Azure Container Instances is the fastest and simplest way to run a container in Azure, without having to provision any virtual machines or adopt a higher-level orchestration service.
Artifact Registry (beta) Container Registry	Azure Container Registry	Allows customers to store Docker formatted images. Used to create all types of container deployments on Azure.
Kubernetes Engine (GKE)	Azure Kubernetes Service (AKS)	Deploy orchestrated containerized applications with Kubernetes. Simplify cluster management and monitoring through automatic upgrades and a built-in operations console. See AKS solution journey.
Kubernetes Engine Monitoring	Azure Monitor for containers	Azure Monitor for containers is a feature designed to monitor the performance of container workloads deployed to: Managed Kubernetes clusters hosted on Azure Kubernetes Service (AKS); Self-managed Kubernetes clusters hosted on Azure using AKS Engine; Azure Container Instances, Self-managed Kubernetes clusters hosted on Azure Stack or on-premises; or Azure Red Hat OpenShift.

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Anthos Service Mesh	Service Fabric Mesh	Fully managed service that enables developers to deploy microservices applications without managing virtual machines, storage, or networking.

Container architectures

Here are some architectures that use AKS as the orchestrator.

- Baseline architecture on Azure Kubernetes Service (AKS)
  - 07/20/2020
  - 37 min read

  Deploy a baseline infrastructure that deploys an AKS cluster with focus on security.
- Microservices architecture on Azure Kubernetes Service (AKS)
  - 5/07/2020
  - 17 min read

  Deploy a microservices architecture on Azure Kubernetes Service (AKS)
- CI/CD pipeline for container-based workloads
  - 7/05/2018
  - 7 min read

  Build a DevOps pipeline for a Node.js web app with Jenkins, Azure Container Registry, Azure Kubernetes Service, Cosmos DB, and Grafana.

view all

Functions

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Functions	Azure Functions	Integrate systems and run backend processes in response to events or schedules without provisioning or managing servers.

Serverless architectures

- Social App for Mobile and Web with Authentication
  - 12/16/2019
  - 3 min read

View a detailed, step-by-step diagram depicting the build process and implementation of the mobile client app architecture that offers social image sharing with a companion web app and authentication abilities, even while offline.

- HIPAA and HITRUST compliant health data AI

  - 12/16/2019
  - 2 min read

  Manage HIPAA and HITRUST compliant health data and medical records with the highest level of built-in security.

- Cross Cloud Scaling Architecture

  - 12/16/2019
  - 1 min read

  Learn how to improve cross cloud scalability with solution architecture that includes Azure Stack. A step-by-step flowchart details instructions for implementation.

DevOps and application monitoring

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Operations (formerly Stackdriver)	Azure Monitor	Maximizes the availability and performance of your applications and services by delivering a comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments. It helps you understand how your applications are performing and proactively identifies issues affecting them and the resources on which they depend.
Cloud Trace	Azure Monitor	Maximizes the availability and performance of your applications and services by delivering a comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments. It helps you understand how your applications are performing and proactively identifies issues affecting them and the resources on which they depend.

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Debugger	Azure Monitor	Maximizes the availability and performance of your applications and services by delivering a comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments. It helps you understand how your applications are performing and proactively identifies issues affecting them and the resources on which they depend.
Cloud Profiler	Azure Monitor	Maximizes the availability and performance of your applications and services by delivering a comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments. It helps you understand how your applications are performing and proactively identifies issues affecting them and the resources on which they depend.
Cloud Source Repositories	Azure Repos, GitHub Repos	A cloud service for collaborating on code development.
Cloud Build	Azure Pipelines, GitHub Actions	Fully managed build service that supports continuous integration and deployment.
Artifact Registry	Azure Artifacts, GitHub Packages	Add fully integrated package management to your continuous integration/continuous delivery (CI/CD) pipelines with a single click. Create and share Maven, npm, NuGet, and Python package feeds from public and private sources with teams of any size.
Cloud Developer Tools (including Cloud Code)	Azure Developer Tools	Collection of tools for building, debugging, deploying, diagnosing, and managing multiplatform scalable apps and services.
Gcloud SDK	Azure CLI	The Azure command-line interface (Azure CLI) is a set of commands used to create and manage Azure resources. The Azure CLI is available across Azure services and is designed to get you working quickly with Azure, with an emphasis on automation.

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Shell	Azure Cloud Shell	Azure Cloud Shell is an interactive, authenticated, browser-accessible shell for managing Azure resources. It provides the flexibility of choosing the shell experience that best suits the way you work, either Bash or PowerShell.
PowerShell on Google Cloud	Azure PowerShell	Azure PowerShell is a set of cmdlets for managing Azure resources directly from the PowerShell command line. Azure PowerShell is designed to make it easy to learn and get started with, but provides powerful features for automation. Written in .NET Standard, Azure PowerShell works with PowerShell 5.1 on Windows, and PowerShell 6.x and higher on all platforms.
Cloud Deployment Manager	Azure Automation	Delivers a cloud-based automation and configuration service that supports consistent management across your Azure and non-Azure environments. It comprises process automation, configuration management, update management, shared capabilities, and heterogeneous features. Automation gives you complete control during deployment, operations, and decommissioning of workloads and resources.
Cloud Deployment Manager	Azure Resource Manager	Provides a way for users to automate the manual, long-running, error-prone, and frequently repeated IT tasks.

DevOps architectures

- Container CI/CD using Jenkins and Kubernetes on Azure Kubernetes Service (AKS)

  - 12/16/2019
  - 2 min read

  Containers make it easy for you to continuously build and deploy applications. By orchestrating the deployment of those containers using Azure Kubernetes Service (AKS), you can achieve replicable, manageable clusters of containers.

- Run a Jenkins server on Azure

  - 11/19/2020
  - 6 min read

  Recommended architecture that shows how to deploy and operate a scalable, enterprise-grade Jenkins

server on Azure secured with single sign-on (SSO).

- DevOps in a hybrid environment
  - 12/16/2019
  - 3 min read

  The tools provided in Azure allow for the implementation of a DevOps strategy that capably manages both cloud and on-premises environments in tandem.

view all

Internet of things (IoT)

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud IoT Core	Azure IoT Hub,Azure Event Hubs	A cloud gateway for managing bidirectional communication with billions of IoT devices, securely and at scale.
Cloud Pub/Sub	Azure Stream Analytics,HDInsight Kafka	Process and route streaming data to a subsequent processing engine or to a storage or database platform.
Edge TPU	Azure IoT Edge	Deploy cloud intelligence directly on IoT devices to run in on-premises scenarios.

IOT architectures

- IoT Architecture ◆ Azure IoT Subsystems
  - 12/16/2019
  - 1 min read

  Learn about our recommended IoT application architecture that supports hybrid cloud and edge computing. A flowchart details how the subsystems function within the IoT application.

- Azure IoT reference architecture
  - 9/10/2020
  - 12 min read

  Recommended architecture for IoT applications on Azure using PaaS (platform-as-a-service) components

- Process real-time vehicle data using IoT
  - 11/17/2020

- 5 min read

This example builds a real-time data ingestion/processing pipeline to ingest and process messages from IoT devices into a big data analytic platform in Azure.

view all

Management

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Billing	Azure Billing API	Services to help generate, monitor, forecast, and share billing data for resource usage by time, organization, or product resources.
Cloud Console	Azure portal	A unified management console that simplifies building, deploying, and operating your cloud resources.
Operations (formerly Stackdriver)	Azure Monitor	Comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments.
Cost Management	Azure Cost Management	Azure Cost Management helps you understand your Azure invoice, manage your billing account and subscriptions, control Azure spending, and optimize resource use.

Messaging and eventing

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Pub/Sub	Azure Service Bus	Supports a set of cloud-based, message-oriented middleware technologies including reliable message queuing and durable publish/subscribe messaging.
Cloud Pub/Sub	Azure Event Grid	A fully managed event routing service that allows for uniform event consumption using a publish/subscribe model.
Cloud Pub/Sub	Azure Event Hubs	A real-time data ingestion and microbatching service used to build dynamic data pipelines and integrates with other Azure services.

Messaging architectures

-

Anomaly Detector Process

- 12/16/2019
- 1 min read

Learn more about Anomaly Detector with a step-by-step flowchart that details the process. See how anomaly detection models are selected with time-series data.

-

Scalable web application

- 10/03/2019
- 7 min read

Use the proven practices in this reference architecture to improve scalability and performance in an Azure App Service web application..

-

Enterprise integration using queues and events

- 12/03/2018
- 5 min read

Recommended architecture for implementing an enterprise integration pattern with Azure Logic Apps, Azure API Management, Azure Service Bus, and Azure Event Grid.

-

Ops automation using Event Grid

- 12/16/2019
- 1 min read

Event Grid allows you to speed automation and simplify policy enforcement. For example, Event Grid can notify Azure Automation when a virtual machine is created, or a SQL Database is spun up. These events can be used to automatically check that service configurations are compliant, put metadata into operations tools, tag virtual machines, or file work items.

Networking

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud virtual networking	Virtual Private Network (VPC)	Azure Virtual Network (Vnet)	Provides an isolated, private environment in the cloud. Users have control over their virtual networking environment, including selection of their own IP address range, adding/updating address ranges, creation of subnets, and configuration of route tables and network gateways.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
DNS management	Cloud DNS	Azure DNS	Manage your DNS records using the same credentials that are used for billing and support contract as your other Azure services
	Cloud DNS	Azure Traffic Manager	Azure Traffic Manager is a DNS-based load balancer that enables you to distribute traffic optimally to services across global Azure regions, while providing high availability and responsiveness.
	Internal DNS	Azure Private DNS	Manages and resolves domain names in the virtual network, without the need to configure a custom DNS solution, and it provides a naming resolution for virtual machines (VMs) within a virtual network and any connected virtual networks.
Hybrid Connectivity	Cloud Interconnect	Azure ExpressRoute	Establishes a private network connection from a location to the cloud provider (not over the Internet).
	Cloud VPN Gateway	Azure Virtual Network Gateway	Connects Azure virtual networks to other Azure virtual networks, or customer on-premises networks (site-to-site). Allows end users to connect to Azure services through VPN tunneling (point-to-site).
	Cloud VPN Gateway	Azure Virtual WAN	Azure virtual WAN simplifies large-scale branch connectivity with VPN and ExpressRoute.
	Cloud router	Azure Virtual Network Gateway	Enables dynamic routes exchange using BGP.
Load balancing	Network Load Balancing	Azure Load Balancer	Azure Load Balancer load-balances traffic at layer 4 (all TCP or UDP).
	Global load balancing	Azure Front door	Azure front door enables global load balancing across regions using a single anycast IP.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
	Global load balancing	Azure Application Gateway	Application Gateway is a layer 7 load balancer. IT takes backends with any IP that is reachable. It supports SSL termination, cookie-based session affinity, and round robin for load-balancing traffic.
	Global load balancing	Azure Traffic Manager	Azure Traffic Manager is a DNS-based load balancer that enables you to distribute traffic optimally to services across global Azure regions, while providing high availability and responsiveness.
Content delivery network	Cloud CDN	Azure CDN	A content delivery network (CDN) is a distributed network of servers that can efficiently deliver web content to users.
Firewall	Firewall rules	Application security groups	Azure Application security groups allow you to group virtual machines and define network security policies based on those groups.
	Firewall rules	Network Security groups	Azure network security group filters network traffic to and from Azure resources in an Azure virtual network.
	Firewall rules	Azure Firewall	Azure Firewall is a managed, cloud-based network security service that protects your Azure Virtual Network resources. It's a fully stateful firewall as a service with built-in high availability and unrestricted cloud scalability.
Web Application Firewall	Cloud Armor	Application Gateway - Web Application Firewall	Azure Web Application Firewall (WAF) provides centralized protection of your web applications from common exploits and vulnerabilities.
	Cloud Armor	Front door – Azure Web Application Firewall	Azure Web Application Firewall (WAF) on Azure Front Door provides centralized protection for your web applications.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
	Cloud Armor	CDN – Azure Web Application Firewall	Azure Web Application Firewall (WAF) on Azure Content Delivery Network (CDN) from Microsoft provides centralized protection for your web content.
NAT Gateway	Cloud NAT	Azure Virtual Network NAT	Virtual Network NAT (network address translation) provides outbound NAT translations for internet connectivity for virtual networks.
Private Connectivity to PaaS	VPC Service controls	Azure Private Link	Azure Private Link enables you to access Azure PaaS Services and Azure hosted customer-owned/partner services over a private endpoint in your virtual network.
Telemetry	VPC Flow logs	NSG Flow logs	Network security group (NSG) flow logs are a feature of Network Watcher that allows you to view information about ingress and egress IP traffic through an NSG.
	Firewall Rules Logging	NSG Flow logs	Network security group (NSG) flow logs are a feature of Network Watcher that allows you to view information about ingress and egress IP traffic through an NSG.
	Operations (formerly Stackdriver)	Azure Monitor	Azure Monitor delivers a comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments. Log queries help you maximize the value of the data collected in Azure Monitor Logs.
	Network Intelligence Center	Azure Network Watcher	Azure Network Watcher provides tools to monitor, diagnose, view metrics, and enable or disable logs for resources in an Azure virtual network.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Other Connectivity Options	S2S,P2S	Direct Interconnect,Partner Interconnect,Carrier Peering	Point to Site lets you create a secure connection to your virtual network from an individual client computer. Site to Site is a connection between two or more networks, such as a corporate network and a branch office network.

Networking architectures

- Deploy highly available NVAs
  - 12/08/2018
  - 7 min read

  Learn how to deploy network virtual appliances for high availability in Azure. This article includes example architectures for ingress, egress, and both.
- Hub-spoke network topology in Azure
  - 9/30/2020
  - 7 min read

  Learn how to implement a hub-spoke topology in Azure, where the hub is a virtual network and the spokes are virtual networks that peer with the hub.
- Implement a secure hybrid network
  - 1/07/2020
  - 9 min read

  See a secure hybrid network that extends an on-premises network to Azure with a perimeter network between the on-premises network and an Azure virtual network.

view all

Security and identity

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Authentication and authorization	Cloud Identity	Azure Active Directory	The Azure Active Directory (Azure AD) enterprise identity service provides single sign-on and multi-factor authentication, which enable the central management of users/groups and external identities federation.
	Identity platform	Azure Active Directory B2C	A highly available and global identity management service for consumer-facing applications, which scales to hundreds of millions of identities. Manage customer, consumer, and citizen access to your business-to-consumer (B2C) applications.
Multi-factor Authentication	Multi-factor Authentication	Azure Active Directory Multi-factor Authentication	Safeguard access to data and applications, while meeting user demand for a simple sign-in process.
RBAC	Identity and Access Management	Azure role-based access control	Azure role-based access control (Azure RBAC) helps you manage who has access to Azure resources, what they can do with those resources, and what areas they have access to.
ABAC	Identity and Access Management	Azure attribute-based access control	Azure attribute-based access control (Azure ABAC) is an authorization system that defines access, based on attributes that are associated with security principals, resources, and environment.
Zero trust	BeyondCorp Enterprise	Azure AD Conditional Access	Conditional Access is the tool used by Azure Active Directory to bring signals together, to make decisions, and to enforce organizational policies.
Resource management	Resource Manager	Azure Resource Manager	Provides a management layer that enables you to create, update, and delete resources in your Azure account, like access control, locks, and tags, to secure and organize your resources after deployment.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Encryption	Cloud KMS, Secret Manager	Azure Key Vault	Provides a security solution and works with other services by allowing you to manage, create, and control encryption keys that are stored in hardware security modules (HSM).
Data-at-rest encryption	Encryption at rest	Azure Storage Service Encryption - encryption by default	Azure Storage Service Encryption helps you protect and safeguard your data and meet your organizational security and compliance commitments.
Data in-use	Confidential Computing	Azure Confidential Computing	Encrypt data in-use.
Hardware security module (HSM)	Cloud HSM	Azure Dedicated HSM	Azure service that provides cryptographic key storage in Azure, to host encryption keys and perform cryptographic operations in a high-availabilty service of FIPS 140-2 Level 3 certified hardware security modules (HSMs).
Data loss prevention (DLP)	Cloud Data Loss Prevention	Azure Information Protection	Azure Information Protection (AIP) is a cloud-based solution that enables organizations to discover, classify, and protect documents and emails by applying labels to content.
Security	Security Command Center, Web Security Scanner	Microsoft Defender for Cloud	An automated security assessment service that improves the security and compliance of applications. Automatically assess applications for vulnerabilities or deviations from best practices.
Threat detection	Event Threat Detection	Azure Advanced Threat Protection	Detect and investigate advanced attacks on-premises and in the cloud.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
SIEM	Chronicle	Microsoft Sentinel	A cloud-native security information and event manager (SIEM) platform that uses built-in AI to help analyze large volumes of data from all sources, including users, applications, servers, and devices that are running on-premises or in any cloud.
Container security	Container Security	Container Security in Microsoft Defender for Cloud	Microsoft Defender for Cloud is the Azure-native solution for securing your containers.
	Artifact Registry	Azure Container Registry	A managed, private Docker registry service that's based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your private Docker container images and related artifacts that allow you to only deploy trusted containers.
	Container Analysis	Microsoft Defender for container registries	Perform vulnerability scans on all container images when they're pushed to the registry, imported into the registry, or pulled within the last 30 days.

Security architectures

- Real-time fraud detection
  - 7/05/2018
  - 4 min read

  Detect fraudulent activity in real-time using Azure Event Hubs and Stream Analytics.
- Securely managed web applications
  - 5/09/2019
  - 8 min read

  Learn about deploying secure applications using the Azure App Service Environment, the Azure Application Gateway service, and Web Application Firewall.
-

Threat indicators for cyber threat intelligence in Azure Sentinel

- 4/13/2020
- 13 min read

Import threat indicators, view logs, create rules to generate security alerts and incidents, and visualize threat intelligence data with Azure Sentinel.

view all

Storage

Object storage

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Cloud Storage Cloud Storage for Firebase	Azure Blob storage	Object storage service, for use cases including cloud applications, content distribution, backup, archiving, disaster recovery, and big data analytics.

Block storage

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Persistant Disk Local SSD	Azure managed disks	SSD storage optimized for I/O intensive read/write operations. For use as high-performance Azure virtual machine storage.

File storage

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Filestore	Azure Files, Azure NetApp Files	File based storage and hosted NetApp Appliance Storage.
Google Drive	OneDrive For business	Cloud storage and file sharing solution for businesses to store, access, and share files anytime and anywhere.

Storage architectures

-

HIPAA and HITRUST compliant health data AI

- 12/16/2019
- 2 min read

Manage HIPAA and HITRUST compliant health data and medical records with the highest level of built-in security.

-

HPC Media Rendering

- 11/04/2020
- 2 min read

Optimize the media rendering process with a step-by-step HPC solution architecture from Azure that combines Azure CycleCloud and HPC Cache.

-

Medical Data Storage Solutions

- 12/16/2019
- 2 min read

Store healthcare data effectively and affordably with cloud-based solutions from Azure. Manage medical records with the highest level of built-in security.

view all

Bulk data transfer

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Transfer Appliance	Azure Import/Export	A data transport solution that uses secure disks and appliances to transfer large amounts of data. Also offers data protection during transit.
Transfer Appliance	Azure Data Box	Petabyte- to exabyte-scale data transport solution that uses secure data storage devices to transfer large amounts of data to and from Azure.

Application services

GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
App Engine	Azure App Service	Managed hosting platform providing easy to use services for deploying and scaling web applications and services.
Apigee	Azure API Management	A turnkey solution for publishing APIs to external and internal consumers.

Web architectures

-

Architect scalable e-commerce web app

- 12/16/2019
- 1 min read

The e-commerce website includes simple order processing workflows with the help of Azure services. Using Azure Functions and Web Apps, developers can focus on building personalized experiences and let Azure take care of the infrastructure.

-

Multi-region N-tier application

- 6/18/2019
- 10 min read

Deploy an application on Azure virtual machines in multiple regions for high availability and resiliency.

-

Serverless web application

- 5/28/2019
- 16 min read

This reference architecture shows a serverless web application, which serves static content from Azure Blob Storage and implements an API using Azure Functions.

view all

Miscellaneous

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Workflow	Composer	Azure Logic Apps	Serverless technology for connecting apps, data and devices anywhere, whether on-premises or in the cloud for large ecosystems of SaaS and cloud-based connectors.
Enterprise application services	G Suite	Microsoft 365	Fully integrated Cloud service providing communications, email, document management in the cloud and available on a wide variety of devices.
Gaming	Game Servers	Azure PlayFab	Managed services for hosting dedicated game servers.
Hybrid	Anthos	Azure Arc	For customers who want to simplify complex and distributed environments across on-premises, edge and multi-cloud, Azure Arc enables deployment of Azure services anywhere and extends Azure management to any infrastructure.
Blockchain	Digital Asset	Azure Confidential Ledger	Tamperproof, unstructured data store hosted in trusted execution environments and backed by cryptographically verifiable evidence.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
Monitoring	Cloud Monitoring	Application Insights	Service that provides visibility into the performance, uptime, and overall health of cloud-powered applications.
Logging	Cloud Logging	Log Analytics	Service for real-time log management and analysis.

Migration tools

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION
App migration to containers	Migrate for Anthos	Azure Migrate: App Containerization tool	Modernize your application by migrating it to AKS or App Services containers.
Migration of virtual machines	Migrate for Compute Engine	Azure Migrate: Server Migration tool	Migrate servers from anywhere to Azure.
VMware migration	Google Cloud VMware Engine	Azure VMware Solution	Move or extend on-premises VMware environments to Azure.
Migration of databases	Database Migration Service	Azure Database Migration Service	Fully managed service designed to enable seamless migrations from multiple database sources to Azure data platforms with minimal downtime.
Migration programs	Google Cloud Rapid Assessment & Migration Program (RAMP)	Azure Migration and Modernization Program	Learn how to move your apps, data, and infrastructure to Azure using a proven cloud migration and modernization approach.
Server assessment		Movere	Increases business intelligence by accurately presenting entire IT environments within a single day.
Database assessment		Data Migration Assistant	It helps pinpoint potential problems blocking migration. It identifies unsupported features, new features that can benefit you after migration, and the right path for database migration.
Web app assessment and migration		Web app migration assistant	Assess on-premises web apps and migrate them to Azure.

AREA	GOOGLE CLOUD SERVICE	AZURE SERVICE	DESCRIPTION

More learning

If you are new to Azure, review the interactive Core Cloud Services - Introduction to Azure module on Microsoft Learn.

Cloud Design Patterns

11/19/2021 • 5 minutes to read • Edit Online

These design patterns are useful for building reliable, scalable, secure applications in the cloud.

Each pattern describes the problem that the pattern addresses, considerations for applying the pattern, and an example based on Microsoft Azure. Most of the patterns include code samples or snippets that show how to implement the pattern on Azure. However, most of the patterns are relevant to any distributed system, whether hosted on Azure or on other cloud platforms.

Challenges in cloud development

	Data Management Data management is the key element of cloud applications, and influences most of the quality attributes. Data is typically hosted in different locations and across multiple servers for reasons such as performance, scalability or availability, and this can present a range of challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations.
	Design and Implementation Good design encompasses factors such as consistency and coherence in component design and deployment, maintainability to simplify administration and development, and reusability to allow components and subsystems to be used in other applications and in other scenarios. Decisions made during the design and implementation phase have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.
	Messaging The distributed nature of cloud applications requires a messaging infrastructure that connects the components and services, ideally in a loosely coupled manner in order to maximize scalability. Asynchronous messaging is widely used, and provides many benefits, but also brings challenges such as the ordering of messages, poison message management, idempotency, and more.

Catalog of patterns

PATTERN	SUMMARY	CATEGORY
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.	Design and Implementation, Operational Excellence

PATTERN	SUMMARY	CATEGORY
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.	Design and Implementation, Operational Excellence
Asynchronous Request-Reply	Decouple backend processing from a frontend host, where backend processing needs to be asynchronous, but the frontend still needs a clear response.	Messaging
Backends for Frontends	Create separate backend services to be consumed by specific frontend applications or interfaces.	Design and Implementation
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.	Reliability
Cache-Aside	Load data on demand into a cache from a data store	Data Management, Performance Efficiency
Choreography	Let each service decide when and how a business operation is processed, instead of depending on a central orchestrator.	Messaging, Performance Efficiency
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.	Reliability
Claim Check	Split a large message into a claim check and a payload to avoid overwhelming a message bus.	Messaging
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation.	Reliability
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel.	Messaging
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit	Design and Implementation
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.	Data Management, Design and Implementation, Performance Efficiency
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.	Reliability, Performance Efficiency

PATTERN	SUMMARY	CATEGORY
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.	Data Management, Performance Efficiency
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.	Design and Implementation, Operational Excellence
Federated Identity	Delegate authentication to an external identity provider.	Security
Gatekeeper	Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them.	Security
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.	Design and Implementation, Operational Excellence
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.	Design and Implementation, Operational Excellence
Gateway Routing	Route requests to multiple services using a single endpoint.	Design and Implementation, Operational Excellence
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.	Reliability, Operational Excellence
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.	Reliability, Operational Excellence
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.	Data Management, Performance Efficiency
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.	Design and Implementation, Reliability
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.	Data Management, Operational Excellence

PATTERN	SUMMARY	CATEGORY
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.	Design and Implementation, Messaging
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.	Messaging, Performance Efficiency
Publisher/Subscriber	Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.	Messaging
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.	Reliability, Messaging, Resiliency, Performance Efficiency
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed.	Reliability
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources.	Messaging, Reliability
Sequential Convoy	Process a set of related messages in a defined order, without blocking processing of other groups of messages.	Messaging
Sharding	Divide a data store into a set of horizontal partitions or shards.	Data Management, Performance Efficiency
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.	Design and Implementation, Operational Excellence
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.	Design and Implementation, Data Management, Performance Efficiency
Strangler Fig	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.	Design and Implementation, Operational Excellence

PATTERN	SUMMARY	CATEGORY
Throttling	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service.	Reliability, Performance Efficiency
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.	Data Management, Security

Data management patterns

11/19/2021 • 2 minutes to read • Edit Online

Data management is the key element of cloud applications, and influences most of the quality attributes. Data is typically hosted in different locations and across multiple servers for reasons such as performance, scalability or availability, and this can present a range of challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations.

Additionally data should be protected at rest, in transit, and via authorized access mechanisms to maintain security assurances of confidentiality, integrity, and availability. Refer to the Azure Security Benchmark Data Protection Control for more information.

PATTERN	SUMMARY
Cache-Aside	Load data on demand into a cache from a data store
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Sharding	Divide a data store into a set of horizontal partitions or shards.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

Design and implementation patterns

11/19/2021 • 2 minutes to read • Edit Online

Good design encompasses factors such as consistency and coherence in component design and deployment, maintainability to simplify administration and development, and reusability to allow components and subsystems to be used in other applications and in other scenarios. Decisions made during the design and implementation phase have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.

PATTERN	SUMMARY
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
Backends for Frontends	Create separate backend services to be consumed by specific frontend applications or interfaces.
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.

PATTERN	SUMMARY
Strangler Fig	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

Messaging patterns

11/19/2021 • 2 minutes to read • Edit Online

The distributed nature of cloud applications requires a messaging infrastructure that connects the components and services, ideally in a loosely coupled manner in order to maximize scalability. Asynchronous messaging is widely used, and provides many benefits, but also brings challenges such as the ordering of messages, poison message management, idempotency, and more.

PATTERN	SUMMARY
Asynchronous Request-Reply	Decouple backend processing from a frontend host, where backend processing needs to be asynchronous, but the frontend still needs a clear response.
Claim Check	Split a large message into a claim check and a payload to avoid overwhelming a message bus.
Choreography	Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control.
Competing Consumers	Enable multiple concurrent consumers to process messages received on the same messaging channel.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Publisher-Subscriber	Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources.
Sequential Convoy	Process a set of related messages in a defined order, without blocking processing of other groups of messages.

Ambassador pattern

11/19/2021 • 3 minutes to read • Edit Online

Create helper services that send network requests on behalf of a consumer service or application. An ambassador service can be thought of as an out-of-process proxy that is co-located with the client.

This pattern can be useful for offloading common client connectivity tasks such as monitoring, logging, routing, security (such as TLS), and resiliency patterns in a language agnostic way. It is often used with legacy applications, or other applications that are difficult to modify, in order to extend their networking capabilities. It can also enable a specialized team to implement those features.

Context and problem

Resilient cloud-based applications require features such as circuit breaking, routing, metering and monitoring, and the ability to make network-related configuration updates. It may be difficult or impossible to update legacy applications or existing code libraries to add these features, because the code is no longer maintained or can't be easily modified by the development team.

Network calls may also require substantial configuration for connection, authentication, and authorization. If these calls are used across multiple applications, built using multiple languages and frameworks, the calls must be configured for each of these instances. In addition, network and security functionality may need to be managed by a central team within your organization. With a large code base, it can be risky for that team to update application code they aren't familiar with.

Solution

Put client frameworks and libraries into an external process that acts as a proxy between your application and external services. Deploy the proxy on the same host environment as your application to allow control over routing, resiliency, security features, and to avoid any host-related access restrictions. You can also use the ambassador pattern to standardize and extend instrumentation. The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application.

Features that are offloaded to the ambassador can be managed independently of the application. You can update and modify the ambassador without disturbing the application's legacy functionality. It also allows for separate, specialized teams to implement and maintain security, networking, or authentication features that have been moved to the ambassador.

Ambassador services can be deployed as a sidecar to accompany the lifecycle of a consuming application or

service. Alternatively, if an ambassador is shared by multiple separate processes on a common host, it can be deployed as a daemon or Windows service. If the consuming service is containerized, the ambassador should be created as a separate container on the same host, with the appropriate links configured for communication.

Issues and considerations

- The proxy adds some latency overhead. Consider whether a client library, invoked directly by the application, is a better approach.
- Consider the possible impact of including generalized features in the proxy. For example, the ambassador could handle retries, but that might not be safe unless all operations are idempotent.
- Consider a mechanism to allow the client to pass some context to the proxy, as well as back to the client. For example, include HTTP request headers to opt out of retry or specify the maximum number of times to retry.
- Consider how you will package and deploy the proxy.
- Consider whether to use a single shared instance for all clients or an instance for each client.

When to use this pattern

Use this pattern when you:

- Need to build a common set of client connectivity features for multiple languages or frameworks.
- Need to offload cross-cutting client connectivity concerns to infrastructure developers or other more specialized teams.
- Need to support cloud or cluster connectivity requirements in a legacy application or an application that is difficult to modify.

This pattern may not be suitable:

- When network request latency is critical. A proxy will introduce some overhead, although minimal, and in some cases this may affect the application.
- When client connectivity features are consumed by a single language. In that case, a better option might be a client library that is distributed to the development teams as a package.
- When connectivity features cannot be generalized and require deeper integration with the client application.

Example

The following diagram shows an application making a request to a remote service via an ambassador proxy. The ambassador provides routing, circuit breaking, and logging. It calls the remote service and then returns the response to the client application:

Related guidance

- Sidecar pattern

Anti-corruption Layer pattern

11/19/2021 • 2 minutes to read • Edit Online

Implement a façade or adapter layer between different subsystems that don't share the same semantics. This layer translates requests that one subsystem makes to the other subsystem. Use this pattern to ensure that an application's design is not limited by dependencies on outside subsystems. This pattern was first described by Eric Evans in *Domain-Driven Design*.

Context and problem

Most applications rely on other systems for some data or functionality. For example, when a legacy application is migrated to a modern system, it may still need existing legacy resources. New features must be able to call the legacy system. This is especially true of gradual migrations, where different features of a larger application are moved to a modern system over time.

Often these legacy systems suffer from quality issues such as convoluted data schemas or obsolete APIs. The features and technologies used in legacy systems can vary widely from more modern systems. To interoperate with the legacy system, the new application may need to support outdated infrastructure, protocols, data models, APIs, or other features that you wouldn't otherwise put into a modern application.

Maintaining access between new and legacy systems can force the new system to adhere to at least some of the legacy system's APIs or other semantics. When these legacy features have quality issues, supporting them "corrupts" what might otherwise be a cleanly designed modern application.

Similar issues can arise with any external system that your development team doesn't control, not just legacy systems.

Solution

Isolate the different subsystems by placing an anti-corruption layer between them. This layer translates communications between the two systems, allowing one system to remain unchanged while the other can avoid compromising its design and technological approach.

The diagram above shows an application with two subsystems. Subsystem A calls to subsystem B through an anti-corruption layer. Communication between subsystem A and the anti-corruption layer always uses the data

model and architecture of subsystem A. Calls from the anti-corruption layer to subsystem B conform to that subsystem's data model or methods. The anti-corruption layer contains all of the logic necessary to translate between the two systems. The layer can be implemented as a component within the application or as an independent service.

Issues and considerations

- The anti-corruption layer may add latency to calls made between the two systems.
- The anti-corruption layer adds an additional service that must be managed and maintained.
- Consider how your anti-corruption layer will scale.
- Consider whether you need more than one anti-corruption layer. You may want to decompose functionality into multiple services using different technologies or languages, or there may be other reasons to partition the anti-corruption layer.
- Consider how the anti-corruption layer will be managed in relation with your other applications or services. How will it be integrated into your monitoring, release, and configuration processes?
- Make sure transaction and data consistency are maintained and can be monitored.
- Consider whether the anti-corruption layer needs to handle all communication between different subsystems, or just a subset of features.
- If the anti-corruption layer is part of an application migration strategy, consider whether it will be permanent, or will be retired after all legacy functionality has been migrated.

When to use this pattern

Use this pattern when:

- A migration is planned to happen over multiple stages, but integration between new and legacy systems needs to be maintained.
- Two or more subsystems have different semantics, but still need to communicate.

This pattern may not be suitable if there are no significant semantic differences between new and legacy systems.

Related guidance

- Strangler Fig pattern

Asynchronous Request-Reply pattern

11/19/2021 • 9 minutes to read • Edit Online

Decouple backend processing from a frontend host, where backend processing needs to be asynchronous, but the frontend still needs a clear response.

Context and problem

In modern application development, it's normal for client applications — often code running in a web-client (browser) — to depend on remote APIs to provide business logic and compose functionality. These APIs may be directly related to the application or may be shared services provided by a third party. Commonly these API calls take place over the HTTP(S) protocol and follow REST semantics.

In most cases, APIs for a client application are designed to respond quickly, on the order of 100 ms or less. Many factors can affect the response latency, including:

- An application's hosting stack.
- Security components.
- The relative geographic location of the caller and the backend.
- Network infrastructure.
- Current load.
- The size of the request payload.
- Processing queue length.
- The time for the backend to process the request.

Any of these factors can add latency to the response. Some can be mitigated by scaling out the backend. Others, such as network infrastructure, are largely out of the control of the application developer. Most APIs can respond quickly enough for responses to arrive back over the same connection. Application code can make a synchronous API call in a non-blocking way, giving the appearance of asynchronous processing, which is recommended for I/O-bound operations.

In some scenarios, however, the work done by backend may be long-running, on the order of seconds, or might be a background process that is executed in minutes or even hours. In that case, it isn't feasible to wait for the work to complete before responding to the request. This situation is a potential problem for any synchronous request-reply pattern.

Some architectures solve this problem by using a message broker to separate the request and response stages. This separation is often achieved by use of the Queue-Based Load Leveling pattern. This separation can allow the client process and the backend API to scale independently. But this separation also brings additional complexity when the client requires success notification, as this step needs to become asynchronous.

Many of the same considerations discussed for client applications also apply for server-to-server REST API calls in distributed systems — for example, in a microservices architecture.

Solution

One solution to this problem is to use HTTP polling. Polling is useful to client-side code, as it can be hard to provide call-back endpoints or use long running connections. Even when callbacks are possible, the extra libraries and services that are required can sometimes add too much extra complexity.

- The client application makes a synchronous call to the API, triggering a long-running operation on the

backend.

- The API responds synchronously as quickly as possible. It returns an HTTP 202 (Accepted) status code, acknowledging that the request has been received for processing.

> **NOTE**
> The API should validate both the request and the action to be performed before starting the long running process. If the request is invalid, reply immediately with an error code such as HTTP 400 (Bad Request).

- The response holds a location reference pointing to an endpoint that the client can poll to check for the result of the long running operation.
- The API offloads processing to another component, such as a message queue.
- While the work is still pending, the status endpoint returns HTTP 202. Once the work is complete, the status endpoint can either return a resource that indicates completion, or redirect to another resource URL. For example, if the asynchronous operation creates a new resource, the status endpoint would redirect to the URL for that resource.

The following diagram shows a typical flow:

1. The client sends a request and receives an HTTP 202 (Accepted) response.
2. The client sends an HTTP GET request to the status endpoint. The work is still pending, so this call also returns HTTP 202.
3. At some point, the work is complete and the status endpoint returns 302 (Found) redirecting to the resource.
4. The client fetches the resource at the specified URL.

Issues and considerations

- There are a number of possible ways to implement this pattern over HTTP and not all upstream services have the same semantics. For example, most services won't return an HTTP 202 response back from a GET method when a remote process hasn't finished. Following pure REST semantics, they should return HTTP 404 (Not Found). This response makes sense when you consider the result of the call isn't present yet.
- An HTTP 202 response should indicate the location and frequency that the client should poll for the response. It should have the following additional headers:

HEADER	DESCRIPTION	NOTES
Location	A URL the client should poll for a response status.	This URL could be a SAS token with the Valet Key Pattern being appropriate if this location needs access control. The valet key pattern is also valid when response polling needs offloading to another backend
Retry-After	An estimate of when processing will complete	This header is designed to prevent polling clients from overwhelming the back-end with retries.

- You may need to use a processing proxy or facade to manipulate the response headers or payload depending on the underlying services used.
- If the status endpoint redirects on completion, either HTTP 302 or HTTP 303 are appropriate return codes, depending on the exact semantics you support.
- Upon successful processing, the resource specified by the Location header should return an appropriate HTTP response code such as 200 (OK), 201 (Created), or 204 (No Content).
- If an error occurs during processing, persist the error at the resource URL described in the Location header and ideally return an appropriate response code to the client from that resource (4xx code).
- Not all solutions will implement this pattern in the same way and some services will include additional or alternate headers. For example, Azure Resource Manager uses a modified variant of this pattern. For more information, see Azure Resource Manager Async Operations.
- Legacy clients might not support this pattern. In that case, you might need to place a facade over the asynchronous API to hide the asynchronous processing from the original client. For example, Azure Logic Apps supports this pattern natively can be used as an integration layer between an asynchronous API and a client that makes synchronous calls. See Perform long-running tasks with the webhook action pattern.
- In some scenarios, you might want to provide a way for clients to cancel a long-running request. In that case, the backend service must support some form of cancellation instruction.

When to use this pattern

Use this pattern for:

- Client-side code, such as browser applications, where it's difficult to provide call-back endpoints, or the use of long-running connections adds too much additional complexity.
- Service calls where only the HTTP protocol is available and the return service can't fire callbacks because of firewall restrictions on the client-side.
- Service calls that need to be integrated with legacy architectures that don't support modern callback technologies such as WebSockets or webhooks.

This pattern might not be suitable when:

- You can use a service built for asynchronous notifications instead, such as Azure Event Grid.
- Responses must stream in real time to the client.
- The client needs to collect many results, and received latency of those results is important. Consider a service bus pattern instead.
- You can use server-side persistent network connections such as WebSockets or SignalR. These services can

be used to notify the caller of the result.

- The network design allows you to open up ports to receive asynchronous callbacks or webhooks.

Example

The following code shows excerpts from an application that uses Azure Functions to implement this pattern. There are three functions in the solution:

- The asynchronous API endpoint.
- The status endpoint.
- A backend function that takes queued work items and executes them.

This sample is available on GitHub.

AsyncProcessingWorkAcceptor function

The `AsyncProcessingWorkAcceptor` function implements an endpoint that accepts work from a client application and puts it on a queue for processing.

- The function generates a request ID and adds it as metadata to the queue message.
- The HTTP response includes a location header pointing to a status endpoint. The request ID is part of the URL path.

```
public static class AsyncProcessingWorkAcceptor
{
    [FunctionName("AsyncProcessingWorkAcceptor")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "post", Route = null)] CustomerPOCO customer,
        [ServiceBus("outqueue", Connection = "ServiceBusConnectionAppSetting")] IAsyncCollector<Message>
OutMessage,
        ILogger log)
    {
        if (String.IsNullOrEmpty(customer.id) || String.IsNullOrEmpty(customer.customername))
        {
            return new BadRequestResult();
        }

        string reqid = Guid.NewGuid().ToString();

        string rqs =
$"http://{Environment.GetEnvironmentVariable("WEBSITE_HOSTNAME")}/api/RequestStatus/{reqid}";

        var messagePayload = JsonConvert.SerializeObject(customer);
        Message m = new Message(Encoding.UTF8.GetBytes(messagePayload));
        m.UserProperties["RequestGUID"] = reqid;
        m.UserProperties["RequestSubmittedAt"] = DateTime.Now;
        m.UserProperties["RequestStatusURL"] = rqs;

        await OutMessage.AddAsync(m);

        return (ActionResult) new AcceptedResult(rqs, $"Request Accepted for
Processing{Environment.NewLine}ProxyStatus: {rqs}");
    }
}
```

AsyncProcessingBackgroundWorker function

The `AsyncProcessingBackgroundWorker` function picks up the operation from the queue, does some work based on the message payload, and writes the result to the SAS signature location.

```
public static class AsyncProcessingBackgroundWorker
{
    [FunctionName("AsyncProcessingBackgroundWorker")]
    public static void Run(
        [ServiceBusTrigger("outqueue", Connection = "ServiceBusConnectionAppSetting")]Message myQueueItem,
        [Blob("data", FileAccess.ReadWrite, Connection = "StorageConnectionAppSetting")] CloudBlobContainer
inputBlob,
        ILogger log)
    {
        // Perform an actual action against the blob data source for the async readers to be able to check
against.
        // This is where your actual service worker processing will be performed.

        var id = myQueueItem.UserProperties["RequestGUID"] as string;

        CloudBlockBlob cbb = inputBlob.GetBlockBlobReference($"{id}.blobdata");

        // Now write the results to blob storage.
        cbb.UploadFromByteArrayAsync(myQueueItem.Body, 0, myQueueItem.Body.Length);
    }
}
```

AsyncOperationStatusChecker function

The `AsyncOperationStatusChecker` function implements the status endpoint. This function first checks whether the request was completed

- If the request was completed, the function either returns a valet-key to the response, or redirects the call immediately to the valet-key URL.
- If the request is still pending, then we should return a 202 accepted with a self-referencing Location header, putting an ETA for a completed response in the http Retry-After header.

```
public static class AsyncOperationStatusChecker
{
    [FunctionName("AsyncOperationStatusChecker")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Anonymous, "get", Route = "RequestStatus/{thisGUID}")] HttpRequest
req,
        [Blob("data/{thisGuid}.blobdata", FileAccess.Read, Connection = "StorageConnectionAppSetting")]
CloudBlockBlob inputBlob, string thisGUID,
        ILogger log)
    {

        OnCompleteEnum OnComplete = Enum.Parse<OnCompleteEnum>(req.Query["OnComplete"].FirstOrDefault() ??
"Redirect");
        OnPendingEnum OnPending = Enum.Parse<OnPendingEnum>(req.Query["OnPending"].FirstOrDefault() ??
"Accepted");

        log.LogInformation($"C# HTTP trigger function processed a request for status on {thisGUID} -
OnComplete {OnComplete} - OnPending {OnPending}");

        // Check to see if the blob is present.
        if (await inputBlob.ExistsAsync())
        {
            // If it's present, depending on the value of the optional "OnComplete" parameter choose what to
do.
            return await OnCompleted(OnComplete, inputBlob, thisGUID);
        }
        else
        {
            // If it's NOT present, check the optional "OnPending" parameter.
            string rqs =
$"http://{Environment.GetEnvironmentVariable("WEBSITE_HOSTNAME")}/api/RequestStatus/{thisGUID}";

            switch (OnPending)
            {
                case OnPendingEnum.Accepted:
                    {
                        // Return an HTTP 202 status code.
                        return (ActionResult)new AcceptedResult() { Location = rqs };
                    }

                case OnPendingEnum.Synchronous:
                    {
                        // Back off and retry. Time out if the backoff period hits one minute
                        int backoff = 250;

                        while (!await inputBlob.ExistsAsync() && backoff < 64000)
                        {
                            log.LogInformation($"Synchronous mode {thisGUID}.blob - retrying in {backoff}
ms");

                            backoff = backoff * 2;
                            await Task.Delay(backoff);
                        }

                        if (await inputBlob.ExistsAsync())
                        {
                            log.LogInformation($"Synchronous Redirect mode {thisGUID}.blob - completed after
{backoff} ms");
                            return await OnCompleted(OnComplete, inputBlob, thisGUID);
                        }
                        else
                        {
                            log.LogInformation($"Synchronous mode {thisGUID}.blob - NOT FOUND after timeout
```

```
                            log.LogInformation($"Synchronous mode {thisGUID}.blob - NOT FOUND after timeout
{backoff} ms");
                            return (ActionResult)new NotFoundResult();
                        }
                    }

                default:
                    {
                        throw new InvalidOperationException($"Unexpected value: {OnPending}");
                    }
            }
        }
    }

    private static async Task<IActionResult> OnCompleted(OnCompleteEnum OnComplete, CloudBlockBlob
inputBlob, string thisGUID)
    {
        switch (OnComplete)
        {
            case OnCompleteEnum.Redirect:
                {
                    // Redirect to the SAS URI to blob storage
                    return (ActionResult)new RedirectResult(inputBlob.GenerateSASURI());
                }

            case OnCompleteEnum.Stream:
                {
                    // Download the file and return it directly to the caller.
                    // For larger files, use a stream to minimize RAM usage.
                    return (ActionResult)new OkObjectResult(await inputBlob.DownloadTextAsync());
                }

            default:
                {
                    throw new InvalidOperationException($"Unexpected value: {OnComplete}");
                }
        }
    }
}

public enum OnCompleteEnum {

    Redirect,
    Stream
}

public enum OnPendingEnum {

    Accepted,
    Synchronous
}
```

Next steps

The following information may be relevant when implementing this pattern:

- Azure Logic Apps - Perform long-running tasks with the polling action pattern.
- For general best practices when designing a web API, see Web API design.

Related guidance

- Backends for Frontends pattern

Backends for Frontends pattern

11/19/2021 • 3 minutes to read • Edit Online

Create separate backend services to be consumed by specific frontend applications or interfaces. This pattern is useful when you want to avoid customizing a single backend for multiple interfaces. This pattern was first described by Sam Newman.

Context and problem

An application may initially be targeted at a desktop web UI. Typically, a backend service is developed in parallel that provides the features needed for that UI. As the application's user base grows, a mobile application is developed that must interact with the same backend. The backend service becomes a general-purpose backend, serving the requirements of both the desktop and mobile interfaces.

But the capabilities of a mobile device differ significantly from a desktop browser, in terms of screen size, performance, and display limitations. As a result, the requirements for a mobile application backend differ from the desktop web UI.

These differences result in competing requirements for the backend. The backend requires regular and significant changes to serve both the desktop web UI and the mobile application. Often, separate interface teams work on each frontend, causing the backend to become a bottleneck in the development process. Conflicting update requirements, and the need to keep the service working for both frontends, can result in spending a lot of effort on a single deployable resource.

As the development activity focuses on the backend service, a separate team may be created to manage and maintain the backend. Ultimately, this results in a disconnect between the interface and backend development teams, placing a burden on the backend team to balance the competing requirements of the different UI teams. When one interface team requires changes to the backend, those changes must be validated with other interface teams before they can be integrated into the backend.

Solution

Create one backend per user interface. Fine-tune the behavior and performance of each backend to best match the needs of the frontend environment, without worrying about affecting other frontend experiences.

Because each backend is specific to one interface, it can be optimized for that interface. As a result, it will be smaller, less complex, and likely faster than a generic backend that tries to satisfy the requirements for all interfaces. Each interface team has autonomy to control their own backend and doesn't rely on a centralized backend development team. This gives the interface team flexibility in language selection, release cadence, prioritization of workload, and feature integration in their backend.

For more information, see Pattern: Backends For Frontends.

Issues and considerations

- Consider how many backends to deploy.
- If different interfaces (such as mobile clients) will make the same requests, consider whether it is necessary to implement a backend for each interface, or if a single backend will suffice.
- Code duplication across services is highly likely when implementing this pattern.
- Frontend-focused backend services should only contain client-specific logic and behavior. General business logic and other global features should be managed elsewhere in your application.
- Think about how this pattern might be reflected in the responsibilities of a development team.
- Consider how long it will take to implement this pattern. Will the effort of building the new backends incur technical debt, while you continue to support the existing generic backend?

When to use this pattern

Use this pattern when:

- A shared or general purpose backend service must be maintained with significant development overhead.
- You want to optimize the backend for the requirements of specific client interfaces.
- Customizations are made to a general-purpose backend to accommodate multiple interfaces.
- An alternative language is better suited for the backend of a different user interface.

This pattern may not be suitable:

- When interfaces make the same or similar requests to the backend.
- When only one interface is used to interact with the backend.

Next steps

- Pattern: Backends For Frontends

Related guidance

- Gateway Aggregation pattern
- Gateway Offloading pattern
- Gateway Routing pattern

Bulkhead pattern

11/19/2021 • 4 minutes to read • Edit Online

The Bulkhead pattern is a type of application design that is tolerant of failure. In a bulkhead architecture, elements of an application are isolated into pools so that if one fails, the others will continue to function. It's named after the sectioned partitions (bulkheads) of a ship's hull. If the hull of a ship is compromised, only the damaged section fills with water, which prevents the ship from sinking.

Context and problem

A cloud-based application may include multiple services, with each service having one or more consumers. Excessive load or failure in a service will impact all consumers of the service.

Moreover, a consumer may send requests to multiple services simultaneously, using resources for each request. When the consumer sends a request to a service that is misconfigured or not responding, the resources used by the client's request may not be freed in a timely manner. As requests to the service continue, those resources may be exhausted. For example, the client's connection pool may be exhausted. At that point, requests by the consumer to other services are affected. Eventually the consumer can no longer send requests to other services, not just the original unresponsive service.

The same issue of resource exhaustion affects services with multiple consumers. A large number of requests originating from one client may exhaust available resources in the service. Other consumers are no longer able to consume the service, causing a cascading failure effect.

Solution

Partition service instances into different groups, based on consumer load and availability requirements. This design helps to isolate failures, and allows you to sustain service functionality for some consumers, even during a failure.

A consumer can also partition resources, to ensure that resources used to call one service don't affect the resources used to call another service. For example, a consumer that calls multiple services may be assigned a connection pool for each service. If a service begins to fail, it only affects the connection pool assigned for that service, allowing the consumer to continue using the other services.

The benefits of this pattern include:

- Isolates consumers and services from cascading failures. An issue affecting a consumer or service can be isolated within its own bulkhead, preventing the entire solution from failing.
- Allows you to preserve some functionality in the event of a service failure. Other services and features of the application will continue to work.
- Allows you to deploy services that offer a different quality of service for consuming applications. A high-priority consumer pool can be configured to use high-priority services.

The following diagram shows bulkheads structured around connection pools that call individual services. If Service A fails or causes some other issue, the connection pool is isolated, so only workloads using the thread pool assigned to Service A are affected. Workloads that use Service B and C are not affected and can continue working without interruption.

The next diagram shows multiple clients calling a single service. Each client is assigned a separate service instance. Client 1 has made too many requests and overwhelmed its instance. Because each service instance is isolated from the others, the other clients can continue making calls.

Issues and considerations

- Define partitions around the business and technical requirements of the application.
- When partitioning services or consumers into bulkheads, consider the level of isolation offered by the technology as well as the overhead in terms of cost, performance and manageability.
- Consider combining bulkheads with retry, circuit breaker, and throttling patterns to provide more sophisticated fault handling.
- When partitioning consumers into bulkheads, consider using processes, thread pools, and semaphores. Projects like resilience4j and Polly offer a framework for creating consumer bulkheads.
- When partitioning services into bulkheads, consider deploying them into separate virtual machines, containers, or processes. Containers offer a good balance of resource isolation with fairly low overhead.
- Services that communicate using asynchronous messages can be isolated through different sets of queues. Each queue can have a dedicated set of instances processing messages on the queue, or a single group of instances using an algorithm to dequeue and dispatch processing.
- Determine the level of granularity for the bulkheads. For example, if you want to distribute tenants across partitions, you could place each tenant into a separate partition, or put several tenants into one partition.
- Monitor each partition's performance and SLA.

When to use this pattern

Use this pattern to:

- Isolate resources used to consume a set of backend services, especially if the application can provide some level of functionality even when one of the services is not responding.
- Isolate critical consumers from standard consumers.
- Protect the application from cascading failures.

This pattern may not be suitable when:

- Less efficient use of resources may not be acceptable in the project.
- The added complexity is not necessary

Example

The following Kubernetes configuration file creates an isolated container to run a single service, with its own CPU and memory resources and limits.

```
apiVersion: v1
kind: Pod
metadata:
  name: drone-management
spec:
  containers:
  - name: drone-management-container
    image: drone-service
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "1"
```

Related guidance

- Designing reliable Azure applications
- Circuit Breaker pattern
- Retry pattern
- Throttling pattern

Cache-Aside pattern

11/19/2021 • 7 minutes to read • Edit Online

Load data on demand into a cache from a data store. This can improve performance and also helps to maintain consistency between data held in the cache and data in the underlying data store.

Context and problem

Applications use a cache to improve repeated access to information held in a data store. However, it's impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is as up-to-date as possible, but can also detect and handle situations that arise when the data in the cache has become stale.

Solution

Many commercial caching systems provide read-through and write-through/write-behind operations. In these systems, an application retrieves data by referencing the cache. If the data isn't in the cache, it's retrieved from the data store and added to the cache. Any modifications to data held in the cache are automatically written back to the data store as well.

For caches that don't provide this functionality, it's the responsibility of the applications that use the cache to maintain the data.

An application can emulate the functionality of read-through caching by implementing the cache-aside strategy. This strategy loads data into the cache on demand. The figure illustrates using the Cache-Aside pattern to store data in the cache.

If an application updates information, it can follow the write-through strategy by making the modification to the data store, and by invalidating the corresponding item in the cache.

When the item is next required, using the cache-aside strategy will cause the updated data to be retrieved from the data store and added back into the cache.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

Lifetime of cached data. Many caches implement an expiration policy that invalidates data and removes it from the cache if it's not accessed for a specified period. For cache-aside to be effective, ensure that the expiration policy matches the pattern of access for applications that use the data. Don't make the expiration period too short because this can cause applications to continually retrieve data from the data store and add it to the cache. Similarly, don't make the expiration period so long that the cached data is likely to become stale. Remember that caching is most effective for relatively static data, or data that is read frequently.

Evicting data. Most caches have a limited size compared to the data store where the data originates, and they'll evict data if necessary. Most caches adopt a least-recently-used policy for selecting items to evict, but this might be customizable. Configure the global expiration property and other properties of the cache, and the expiration property of each cached item, to ensure that the cache is cost effective. It isn't always appropriate to apply a global eviction policy to every item in the cache. For example, if a cached item is very expensive to retrieve from the data store, it can be beneficial to keep this item in the cache at the expense of more frequently accessed but less costly items.

Priming the cache. Many solutions prepopulate the cache with the data that an application is likely to need as part of the startup processing. The Cache-Aside pattern can still be useful if some of this data expires or is evicted.

Consistency. Implementing the Cache-Aside pattern doesn't guarantee consistency between the data store and the cache. An item in the data store can be changed at any time by an external process, and this change might not be reflected in the cache until the next time the item is loaded. In a system that replicates data across data stores, this problem can become serious if synchronization occurs frequently.

Local (in-memory) caching. A cache could be local to an application instance and stored in-memory. Cache-aside can be useful in this environment if an application repeatedly accesses the same data. However, a local cache is private and so different application instances could each have a copy of the same cached data. This data could quickly become inconsistent between caches, so it might be necessary to expire data held in a private cache and refresh it more frequently. In these scenarios, consider investigating the use of a shared or a distributed caching mechanism.

When to use this pattern

Use this pattern when:

- A cache doesn't provide native read-through and write-through operations.
- Resource demand is unpredictable. This pattern enables applications to load data on demand. It makes no assumptions about which data an application will require in advance.

This pattern might not be suitable:

- When the cached data set is static. If the data will fit into the available cache space, prime the cache with the data on startup and apply a policy that prevents the data from expiring.
- For caching session state information in a web application hosted in a web farm. In this environment, you should avoid introducing dependencies based on client-server affinity.

Example

In Microsoft Azure you can use Azure Cache for Redis to create a distributed cache that can be shared by multiple instances of an application.

This following code examples use the StackExchange.Redis client, which is a Redis client library written for .NET.

To connect to an Azure Cache for Redis instance, call the static `ConnectionMultiplexer.Connect` method and pass in the connection string. The method returns a `ConnectionMultiplexer` that represents the connection. One approach to sharing a `ConnectionMultiplexer` instance in your application is to have a static property that returns a connected instance, similar to the following example. This approach provides a thread-safe way to initialize only a single connected instance.

```
private static ConnectionMultiplexer Connection;

// Redis connection string information
private static Lazy<ConnectionMultiplexer> lazyConnection = new Lazy<ConnectionMultiplexer>(() =>
{
    string cacheConnection = ConfigurationManager.AppSettings["CacheConnection"].ToString();
    return ConnectionMultiplexer.Connect(cacheConnection);
});

public static ConnectionMultiplexer Connection => lazyConnection.Value;
```

The `GetMyEntityAsync` method in the following code example shows an implementation of the Cache-Aside pattern. This method retrieves an object from the cache using the read-through approach.

An object is identified by using an integer ID as the key. The `GetMyEntityAsync` method tries to retrieve an item with this key from the cache. If a matching item is found, it's returned. If there's no match in the cache, the `GetMyEntityAsync` method retrieves the object from a data store, adds it to the cache, and then returns it. The code that actually reads the data from the data store is not shown here, because it depends on the data store. Note that the cached item is configured to expire to prevent it from becoming stale if it's updated elsewhere.

```
// Set five minute expiration as a default
private const double DefaultExpirationTimeInMinutes = 5.0;

public async Task<MyEntity> GetMyEntityAsync(int id)
{
  // Define a unique key for this method and its parameters.
  var key = $"MyEntity:{id}";
  var cache = Connection.GetDatabase();

  // Try to get the entity from the cache.
  var json = await cache.StringGetAsync(key).ConfigureAwait(false);
  var value = string.IsNullOrWhiteSpace(json)
                ? default(MyEntity)
                : JsonConvert.DeserializeObject<MyEntity>(json);

  if (value == null) // Cache miss
  {
    // If there's a cache miss, get the entity from the original store and cache it.
    // Code has been omitted because it is data store dependent.
    value = ...;

    // Avoid caching a null value.
    if (value != null)
    {
      // Put the item in the cache with a custom expiration time that
      // depends on how critical it is to have stale data.
      await cache.StringSetAsync(key, JsonConvert.SerializeObject(value)).ConfigureAwait(false);
      await cache.KeyExpireAsync(key,
TimeSpan.FromMinutes(DefaultExpirationTimeInMinutes)).ConfigureAwait(false);
    }
  }

  return value;
}
```

> The examples use Azure Cache for Redis to access the store and retrieve information from the cache. For more information, see Using Azure Cache for Redis and How to create a Web App with Azure Cache for Redis.

The `UpdateEntityAsync` method shown below demonstrates how to invalidate an object in the cache when the value is changed by the application. The code updates the original data store and then removes the cached item from the cache.

```
public async Task UpdateEntityAsync(MyEntity entity)
{
    // Update the object in the original data store.
    await this.store.UpdateEntityAsync(entity).ConfigureAwait(false);

    // Invalidate the current cache object.
    var cache = Connection.GetDatabase();
    var id = entity.Id;
    var key = $"MyEntity:{id}"; // The key for the cached object.
    await cache.KeyDeleteAsync(key).ConfigureAwait(false); // Delete this key from the cache.
}
```

> **NOTE**
>
> The order of the steps is important. Update the data store *before* removing the item from the cache. If you remove the cached item first, there is a small window of time when a client might fetch the item before the data store is updated. That will result in a cache miss (because the item was removed from the cache), causing the earlier version of the item to be fetched from the data store and added back into the cache. The result will be stale cache data.

Related guidance

The following information may be relevant when implementing this pattern:

- Caching Guidance. Provides additional information on how you can cache data in a cloud solution, and the issues that you should consider when you implement a cache.
- Data Consistency Primer. Cloud applications typically use data that's spread across data stores. Managing and maintaining data consistency in this environment is a critical aspect of the system, particularly the concurrency and availability issues that can arise. This primer describes issues about consistency across distributed data, and summarizes how an application can implement eventual consistency to maintain the availability of data.

Choreography pattern

11/19/2021 • 7 minutes to read • Edit Online

Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control.

Context and problem

In microservices architecture, it's often the case that a cloud-based application is divided into several small services that work together to process a business transaction end-to-end. To lower coupling between services, each service is responsible for a single business operation. Some benefits include faster development, smaller code base, and scalability. However, designing an efficient and scalable workflow is a challenge and often requires complex interservice communication.

The services communicate with each other by using well-defined APIs. Even a single business operation can result in multiple point-to-point calls among all services. A common pattern for communication is to use a centralized service that acts as the orchestrator. It acknowledges all incoming requests and delegates operations to the respective services. In doing so, it also manages the workflow of the entire business transaction. Each service just completes an operation and is not aware of the overall workflow.

The orchestrator pattern reduces point-to-point communication between services but has some drawbacks because of the tight coupling between the orchestrator and other services that participate in processing of the business transaction. To execute tasks in a sequence, the orchestrator needs to have some domain knowledge about the responsibilities of those services. If you want to add or remove services, existing logic will break, and you'll need to rewire portions of the communication path. While you can configure the workflow, add or remove services easily with a well-designed orchestrator, such an implementation is complex and hard to maintain.

Solution

Let each service decide when and how a business operation is processed, instead of depending on a central orchestrator.

One way to implement choreography is to use the asynchronous messaging pattern to coordinate the business operations.

A client request publishes messages to a message queue. As messages arrive, they are pushed to subscribers, or services, interested in that message. Each subscribed service does their operation as indicated by the message and responds to the message queue with success or failure of the operation. In case of success, the service can push a message back to the same queue or a different message queue so that another service can continue the workflow if needed. If an operation fails, the message bus can retry that operation.

This way, the services choreograph the workflow among themselves without depending on an orchestrator or having direct communication between them.

Because there isn't point-to-point communication, this pattern helps reduce coupling between services. Also, it can remove the performance bottleneck caused by the orchestrator when it has to deal with all transactions.

When to use this pattern

Use the choreography pattern if you expect to update, remove, or add new services frequently. The entire app can be modified with lesser effort and minimal disruption to existing services.

Consider this pattern if you experience performance bottlenecks in the central orchestrator.

This pattern is a natural model for the serverless architecture where all services can be short lived, or event driven. Services can spin up because of an event, do their task, and are removed when the task is finished.

Issues and considerations

Decentralizing the orchestrator can cause issues while managing the workflow.

If a service fails to complete a business operation, it can be difficult to recover from that failure. One way is to have the service indicate failure by firing an event. Another service subscribes to those failed events takes necessary actions such as applying compensating transactions to undo successful operations in a request. The failed service might also fail to fire an event for the failure. In that case, consider using a retry and, or time out mechanism to recognize that operation as a failure. For an example, see the Example section.

It's simple to implement a workflow when you want to process independent business operations in parallel. You can use a single message bus. However, the workflow can become complicated when choreography needs to occur in a sequence. For instance, Service C can start its operation only after Service A and Service B have completed their operations with success. One approach is to have multiple message buses that get messages in the required order. For more information, see the Example section.

The choreography pattern becomes a challenge if the number of services grow rapidly. Given the high number of independent moving parts, the workflow between services tends to get complex. Also, distributed tracing

becomes difficult.

The orchestrator centrally manages the resiliency of the workflow and it can become a single point of failure. On the other hand, for choreography, the role is distributed between all services and resiliency becomes less robust.

Each service isn't only responsible for the resiliency of its operation but also the workflow. This responsibility can be burdensome for the service and hard to implement. Each service must retry transient, nontransient, and time-out failures, so that the request terminates gracefully, if needed. Also, the service must be diligent about communicating the success or failure of the operation so that other services can act accordingly.

Example

This example shows the choreography pattern with the Drone Delivery app. When a client requests a pickup, the app assigns a drone and notifies the client.

An example of this pattern is available on GitHub.

A single client business transaction requires three distinct business operations: creating or updating a package, assigning a drone to deliver the package, and checking the delivery status. Those operations are performed by three microservices: Package, Drone Scheduler, and Delivery services. Instead of a central orchestrator, the services use messaging to collaborate and coordinate the request among themselves.

Design

The business transaction is processed in a sequence through multiple hops. Each hop has a message bus and the respective business service.

When a client sends a delivery request through an HTTP endpoint, the Ingestion service receives it, raises an operation event, and sends it to a message bus. The bus invokes the subscribed business service and sends the

event in a POST request. On receiving the event, the business service can complete the operation with success, failure, or the request can time out. If successful, the service responds to the bus with the Ok status code, raises a new operation event, and sends it to the message bus of the next hop. In case of a failure or time-out, the service reports failure by sending the BadRequest code to the message bus that sent the original POST request. The message bus retries the operation based on a retry policy. After that period expires, message bus flags the failed operation and further processing of the entire request stops.

This workflow continues until the entire request has been processed.

The design uses multiple message buses to process the entire business transaction. Microsoft Azure Event Grid provides the messaging service. The app is deployed in an Azure Kubernetes Service (AKS) cluster with two containers in the same pod. One container runs the ambassador that interacts with Event Grid while the other runs a business service. The approach with two containers in the same pod improves performance and scalability. The ambassador and the business service share the same network allowing for low latency and high throughput.

To avoid cascading retry operations that might lead to multiple efforts, only Event Grid retries an operation instead of the business service. It flags a failed request by sending a messaging to a dead letter queue (DLQ).

The business services are idempotent to make sure retry operations don't result in duplicate resources. For example, the Package service uses upsert operations to add data to the data store.

The example implements a custom solution to correlate calls across all services and Event Grid hops.

Here's a code example that shows the choreography pattern between all business services. It shows the workflow of the Drone Delivery app transactions. Code for exception handling and logging have been removed for brevity.

```
[HttpPost]
[Route("/api/[controller]/operation")]
[ProducesResponseType(typeof(void), 200)]
[ProducesResponseType(typeof(void), 400)]
[ProducesResponseType(typeof(void), 500)]

public async Task<IActionResult> Post([FromBody] EventGridEvent[] events)
{

   if (events == null)
   {
       return BadRequest("No Event for Choreography");
   }

   foreach(var e in events)
   {

        List<EventGridEvent> listEvents = new List<EventGridEvent>();
        e.Topic = eventRepository.GetTopic();
        e.EventTime = DateTime.Now;
        switch (e.EventType)
        {
            case Operations.ChoreographyOperation.ScheduleDelivery:
            {
                var packageGen = await
packageServiceCaller.UpsertPackageAsync(delivery.PackageInfo).ConfigureAwait(false);
                if (packageGen is null)
                {
                    //BadRequest allows the event to be reprocessed by Event Grid
                    return BadRequest("Package creation failed.");
                }

                //we set the event type to the next choreography step
                e.EventType = Operations.ChoreographyOperation.CreatePackage;
                listEvents.Add(e);
                await eventRepository.SendEventAsync(listEvents);
                return Ok("Created Package Completed");
            }
            case Operations.ChoreographyOperation.CreatePackage:
            {
                var droneId = await
droneSchedulerServiceCaller.GetDroneIdAsync(delivery).ConfigureAwait(false);
                if (droneId is null)
                {
                    //BadRequest allows the event to be reprocessed by Event Grid
                    return BadRequest("could not get a drone id");
                }
                e.Subject = droneId;
                e.EventType = Operations.ChoreographyOperation.GetDrone;
                listEvents.Add(e);
                await eventRepository.SendEventAsync(listEvents);
                return Ok("Drone Completed");
            }
            case Operations.ChoreographyOperation.GetDrone:
            {
                var deliverySchedule = await deliveryServiceCaller.ScheduleDeliveryAsync(delivery,
e.Subject);
                return Ok("Delivery Completed");
            }
            return BadRequest();
    }
}
```

Related guidance

Consider these patterns in your design for choreography.

- Modularize the business service by using the ambassador design pattern.
- Implement queue-based load leveling pattern to handle spikes of the workload.
- Use asynchronous distributed messaging through the publisher-subscriber pattern.
- Use compensating transactions to undo a series of successful operations in case one or more related operation fails.
- For information about using a message broker in a messaging infrastructure, see Asynchronous messaging options in Azure.

Circuit Breaker pattern

11/19/2021 • 17 minutes to read • Edit Online

Handle faults that might take a variable amount of time to recover from, when connecting to a remote service or resource. This can improve the stability and resiliency of an application.

Context and problem

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them by using a strategy such as the Retry pattern.

However, there can also be situations where faults are due to unanticipated events, and that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure in one part of the system might lead to cascading failures. For example, an operation that invokes a service could be configured to implement a timeout, and reply with a failure message if the service fails to respond within this period. However, this strategy could cause many concurrent requests to the same operation to be blocked until the timeout period expires. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on. Consequently, these resources could become exhausted, causing failure of other possibly unrelated parts of the system that need to use the same resources. In these situations, it would be preferable for the operation to fail immediately, and only attempt to invoke the service if it's likely to succeed. Note that setting a shorter timeout might help to resolve this problem, but the timeout shouldn't be so short that the operation fails most of the time, even if the request to the service would eventually succeed.

Solution

The Circuit Breaker pattern, popularized by Michael Nygard in his book, Release It!, can prevent an application from repeatedly trying to execute an operation that's likely to fail. Allowing it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long lasting. The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.

> The purpose of the Circuit Breaker pattern is different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it'll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.

A circuit breaker acts as a proxy for operations that might fail. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.

The proxy can be implemented as a state machine with the following states that mimic the functionality of an

electrical circuit breaker:

- **Closed**: The request from the application is routed to the operation. The proxy maintains a count of the number of recent failures, and if the call to the operation is unsuccessful the proxy increments this count. If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the **Open** state. At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the **Half-Open** state.

  > The purpose of the timeout timer is to give the system time to fix the problem that caused the failure before allowing the application to try to perform the operation again.

- **Open**: The request from the application fails immediately and an exception is returned to the application.
- **Half-Open**: A limited number of requests from the application are allowed to pass through and invoke the operation. If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state (the failure counter is reset). If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure.

  > The **Half-Open** state is useful to prevent a recovering service from suddenly being flooded with requests. As a service recovers, it might be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work can cause the service to time out or fail again.

In the figure, the failure counter used by the **Closed** state is time based. It's automatically reset at periodic intervals. This helps to prevent the circuit breaker from entering the **Open** state if it experiences occasional failures. The failure threshold that trips the circuit breaker into the **Open** state is only reached when a specified number of failures have occurred during a specified interval. The counter used by the **Half-Open** state records the number of successful attempts to invoke the operation. The circuit breaker reverts to the **Closed** state after

a specified number of consecutive operation invocations have been successful. If any invocation fails, the circuit breaker enters the **Open** state immediately and the success counter will be reset the next time it enters the **Half-Open** state.

> How the system recovers is handled externally, possibly by restoring or restarting a failed component or repairing a network connection.

The Circuit Breaker pattern provides stability while the system recovers from a failure and minimizes the impact on performance. It can help to maintain the response time of the system by quickly rejecting a request for an operation that's likely to fail, rather than waiting for the operation to time out, or never return. If the circuit breaker raises an event each time it changes state, this information can be used to monitor the health of the part of the system protected by the circuit breaker, or to alert an administrator when a circuit breaker trips to the **Open** state.

The pattern is customizable and can be adapted according to the type of the possible failure. For example, you can apply an increasing timeout timer to a circuit breaker. You could place the circuit breaker in the **Open** state for a few seconds initially, and then if the failure hasn't been resolved increase the timeout to a few minutes, and so on. In some cases, rather than the **Open** state returning failure and raising an exception, it could be useful to return a default value that is meaningful to the application.

Issues and considerations

You should consider the following points when deciding how to implement this pattern:

Exception Handling. An application invoking an operation through a circuit breaker must be prepared to handle the exceptions raised if the operation is unavailable. The way exceptions are handled will be application specific. For example, an application could temporarily degrade its functionality, invoke an alternative operation to try to perform the same task or obtain the same data, or report the exception to the user and ask them to try again later.

Types of Exceptions. A request might fail for many reasons, some of which might indicate a more severe type of failure than others. For example, a request might fail because a remote service has crashed and will take several minutes to recover, or because of a timeout due to the service being temporarily overloaded. A circuit breaker might be able to examine the types of exceptions that occur and adjust its strategy depending on the nature of these exceptions. For example, it might require a larger number of timeout exceptions to trip the circuit breaker to the **Open** state compared to the number of failures due to the service being completely unavailable.

Logging. A circuit breaker should log all failed requests (and possibly successful requests) to enable an administrator to monitor the health of the operation.

Recoverability. You should configure the circuit breaker to match the likely recovery pattern of the operation it's protecting. For example, if the circuit breaker remains in the **Open** state for a long period, it could raise exceptions even if the reason for the failure has been resolved. Similarly, a circuit breaker could fluctuate and reduce the response times of applications if it switches from the **Open** state to the **Half-Open** state too quickly.

Testing Failed Operations. In the **Open** state, rather than using a timer to determine when to switch to the **Half-Open** state, a circuit breaker can instead periodically ping the remote service or resource to determine whether it's become available again. This ping could take the form of an attempt to invoke an operation that had previously failed, or it could use a special operation provided by the remote service specifically for testing the health of the service, as described by the Health Endpoint Monitoring pattern.

Manual Override. In a system where the recovery time for a failing operation is extremely variable, it's beneficial to provide a manual reset option that enables an administrator to close a circuit breaker (and reset the failure counter). Similarly, an administrator could force a circuit breaker into the **Open** state (and restart the timeout timer) if the operation protected by the circuit breaker is temporarily unavailable.

Concurrency. The same circuit breaker could be accessed by a large number of concurrent instances of an application. The implementation shouldn't block concurrent requests or add excessive overhead to each call to an operation.

Resource Differentiation. Be careful when using a single circuit breaker for one type of resource if there might be multiple underlying independent providers. For example, in a data store that contains multiple shards, one shard might be fully accessible while another is experiencing a temporary issue. If the error responses in these scenarios are merged, an application might try to access some shards even when failure is highly likely, while access to other shards might be blocked even though it's likely to succeed.

Accelerated Circuit Breaking. Sometimes a failure response can contain enough information for the circuit breaker to trip immediately and stay tripped for a minimum amount of time. For example, the error response from a shared resource that's overloaded could indicate that an immediate retry isn't recommended and that the application should instead try again in a few minutes.

> **NOTE**
>
> A service can return HTTP 429 (Too Many Requests) if it is throttling the client, or HTTP 503 (Service Unavailable) if the service is not currently available. The response can include additional information, such as the anticipated duration of the delay.

Replaying Failed Requests. In the **Open** state, rather than simply failing quickly, a circuit breaker could also record the details of each request to a journal and arrange for these requests to be replayed when the remote resource or service becomes available.

Inappropriate Timeouts on External Services. A circuit breaker might not be able to fully protect applications from operations that fail in external services that are configured with a lengthy timeout period. If the timeout is too long, a thread running a circuit breaker might be blocked for an extended period before the circuit breaker indicates that the operation has failed. In this time, many other application instances might also try to invoke the service through the circuit breaker and tie up a significant number of threads before they all fail.

When to use this pattern

Use this pattern:

- To prevent an application from trying to invoke a remote service or access a shared resource if this operation is highly likely to fail.

This pattern isn't recommended:

- For handling access to local private resources in an application, such as in-memory data structure. In this environment, using a circuit breaker would add overhead to your system.
- As a substitute for handling exceptions in the business logic of your applications.

Example

In a web application, several of the pages are populated with data retrieved from an external service. If the system implements minimal caching, most hits to these pages will cause a round trip to the service. Connections from the web application to the service could be configured with a timeout period (typically 60 seconds), and if the service doesn't respond in this time the logic in each web page will assume that the service is unavailable and throw an exception.

However, if the service fails and the system is very busy, users could be forced to wait for up to 60 seconds before an exception occurs. Eventually resources such as memory, connections, and threads could be exhausted,

preventing other users from connecting to the system, even if they aren't accessing pages that retrieve data from the service.

Scaling the system by adding further web servers and implementing load balancing might delay when resources become exhausted, but it won't resolve the issue because user requests will still be unresponsive and all web servers could still eventually run out of resources.

Wrapping the logic that connects to the service and retrieves the data in a circuit breaker could help to solve this problem and handle the service failure more elegantly. User requests will still fail, but they'll fail more quickly and the resources won't be blocked.

The `CircuitBreaker` class maintains state information about a circuit breaker in an object that implements the `ICircuitBreakerStateStore` interface shown in the following code.

```
interface ICircuitBreakerStateStore
{
  CircuitBreakerStateEnum State { get; }

  Exception LastException { get; }

  DateTime LastStateChangedDateUtc { get; }

  void Trip(Exception ex);

  void Reset();

  void HalfOpen();

  bool IsClosed { get; }
}
```

The `State` property indicates the current state of the circuit breaker, and will be either **Open**, **HalfOpen**, or **Closed** as defined by the `CircuitBreakerStateEnum` enumeration. The `IsClosed` property should be true if the circuit breaker is closed, but false if it's open or half open. The `Trip` method switches the state of the circuit breaker to the open state and records the exception that caused the change in state, together with the date and time that the exception occurred. The `LastException` and the `LastStateChangedDateUtc` properties return this information. The `Reset` method closes the circuit breaker, and the `HalfOpen` method sets the circuit breaker to half open.

The `InMemoryCircuitBreakerStateStore` class in the example contains an implementation of the `ICircuitBreakerStateStore` interface. The `CircuitBreaker` class creates an instance of this class to hold the state of the circuit breaker.

The `ExecuteAction` method in the `CircuitBreaker` class wraps an operation, specified as an `Action` delegate. If the circuit breaker is closed, `ExecuteAction` invokes the `Action` delegate. If the operation fails, an exception handler calls `TrackException`, which sets the circuit breaker state to open. The following code example highlights this flow.

```
public class CircuitBreaker
{
  private readonly ICircuitBreakerStateStore stateStore =
    CircuitBreakerStateStoreFactory.GetCircuitBreakerStateStore();

  private readonly object halfOpenSyncObject = new object ();
  ...
  public bool IsClosed { get { return stateStore.IsClosed; } }

  public bool IsOpen { get { return !IsClosed; } }

  public void ExecuteAction(Action action)
  {
    ...
    if (IsOpen)
    {
      // The circuit breaker is Open.
      ... (see code sample below for details)
    }

    // The circuit breaker is Closed, execute the action.
    try
    {
      action();
    }
    catch (Exception ex)
    {
      // If an exception still occurs here, simply
      // retrip the breaker immediately.
      this.TrackException(ex);

      // Throw the exception so that the caller can tell
      // the type of exception that was thrown.
      throw;
    }
  }

  private void TrackException(Exception ex)
  {
    // For simplicity in this example, open the circuit breaker on the first exception.
    // In reality this would be more complex. A certain type of exception, such as one
    // that indicates a service is offline, might trip the circuit breaker immediately.
    // Alternatively it might count exceptions locally or across multiple instances and
    // use this value over time, or the exception/success ratio based on the exception
    // types, to open the circuit breaker.
    this.stateStore.Trip(ex);
  }
}
```

The following example shows the code (omitted from the previous example) that is executed if the circuit breaker isn't closed. It first checks if the circuit breaker has been open for a period longer than the time specified by the local `OpenToHalfOpenWaitTime` field in the `CircuitBreaker` class. If this is the case, the `ExecuteAction` method sets the circuit breaker to half open, then tries to perform the operation specified by the `Action` delegate.

If the operation is successful, the circuit breaker is reset to the closed state. If the operation fails, it is tripped back to the open state and the time the exception occurred is updated so that the circuit breaker will wait for a further period before trying to perform the operation again.

If the circuit breaker has only been open for a short time, less than the `OpenToHalfOpenWaitTime` value, the `ExecuteAction` method simply throws a `CircuitBreakerOpenException` exception and returns the error that caused the circuit breaker to transition to the open state.

Additionally, it uses a lock to prevent the circuit breaker from trying to perform concurrent calls to the operation

while it's half open. A concurrent attempt to invoke the operation will be handled as if the circuit breaker was open, and it'll fail with an exception as described later.

```
...
if (IsOpen)
{
  // The circuit breaker is Open. Check if the Open timeout has expired.
  // If it has, set the state to HalfOpen. Another approach might be to
  // check for the HalfOpen state that had be set by some other operation.
  if (stateStore.LastStateChangedDateUtc + OpenToHalfOpenWaitTime < DateTime.UtcNow)
  {
    // The Open timeout has expired. Allow one operation to execute. Note that, in
    // this example, the circuit breaker is set to HalfOpen after being
    // in the Open state for some period of time. An alternative would be to set
    // this using some other approach such as a timer, test method, manually, and
    // so on, and check the state here to determine how to handle execution
    // of the action.
    // Limit the number of threads to be executed when the breaker is HalfOpen.
    // An alternative would be to use a more complex approach to determine which
    // threads or how many are allowed to execute, or to execute a simple test
    // method instead.
    bool lockTaken = false;
    try
    {
      Monitor.TryEnter(halfOpenSyncObject, ref lockTaken);
      if (lockTaken)
      {
        // Set the circuit breaker state to HalfOpen.
        stateStore.HalfOpen();

        // Attempt the operation.
        action();

        // If this action succeeds, reset the state and allow other operations.
        // In reality, instead of immediately returning to the Closed state, a counter
        // here would record the number of successful operations and return the
        // circuit breaker to the Closed state only after a specified number succeed.
        this.stateStore.Reset();
        return;
      }
    }
    catch (Exception ex)
    {
      // If there's still an exception, trip the breaker again immediately.
      this.stateStore.Trip(ex);

      // Throw the exception so that the caller knows which exception occurred.
      throw;
    }
    finally
    {
      if (lockTaken)
      {
        Monitor.Exit(halfOpenSyncObject);
      }
    }
  }
  // The Open timeout hasn't yet expired. Throw a CircuitBreakerOpen exception to
  // inform the caller that the call was not actually attempted,
  // and return the most recent exception received.
  throw new CircuitBreakerOpenException(stateStore.LastException);
}
...
```

To use a `CircuitBreaker` object to protect an operation, an application creates an instance of the `CircuitBreaker`

class and invokes the `ExecuteAction` method, specifying the operation to be performed as the parameter. The application should be prepared to catch the `CircuitBreakerOpenException` exception if the operation fails because the circuit breaker is open. The following code shows an example:

```
var breaker = new CircuitBreaker();

try
{
  breaker.ExecuteAction(() =>
  {
    // Operation protected by the circuit breaker.
    ...
  });
}
catch (CircuitBreakerOpenException ex)
{
  // Perform some different action when the breaker is open.
  // Last exception details are in the inner exception.
  ...
}
catch (Exception ex)
{
  ...
}
```

Related guidance

The following patterns might also be useful when implementing this pattern:

- Retry pattern. Describes how an application can handle anticipated temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that has previously failed.
- Health Endpoint Monitoring pattern. A circuit breaker might be able to test the health of a service by sending a request to an endpoint exposed by the service. The service should return information indicating its status.

Claim-Check pattern

11/19/2021 • 6 minutes to read • Edit Online

Split a large message into a claim check and a payload. Send the claim check to the messaging platform and store the payload to an external service. This pattern allows large messages to be processed, while protecting the message bus and the client from being overwhelmed or slowed down. This pattern also helps to reduce costs, as storage is usually cheaper than resource units used by the messaging platform.

This pattern is also known as Reference-Based Messaging, and was originally described in the book *Enterprise Integration Patterns*, by Gregor Hohpe and Bobby Woolf.

Context and problem

A messaging-based architecture at some point must be able to send, receive, and manipulate large messages. Such messages may contain anything, including images (for example, MRI scans), sound files (for example, call-center calls), text documents, or any kind of binary data of arbitrary size.

Sending such large messages to the message bus directly is not recommended, because they require more resources and bandwidth to be consumed. Large messages can also slow down the entire solution, because messaging platforms are usually fine-tuned to handle huge quantities of small messages. Also, most messaging platforms have limits on message size, so you may need to work around these limits for large messages.

Solution

Store the entire message payload into an external service, such as a database. Get the reference to the stored payload, and send just that reference to the message bus. The reference acts like a claim check used to retrieve a piece of luggage, hence the name of the pattern. Clients interested in processing that specific message can use the obtained reference to retrieve the payload, if needed.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Consider deleting the message data after consuming it, if you don't need to archive the messages. Although blob storage is relatively cheap, it costs some money in the long run, especially if there is a lot of data. Deleting the message can be done synchronously by the application that receives and processes the message, or asynchronously by a separate dedicated process. The asynchronous approach removes old data with no impact on the throughput and message processing performance of the receiving application.

- Storing and retrieving the message causes some additional overhead and latency. You may want to implement logic in the sending application to use this pattern only when the message size exceeds the data limit of the message bus. The pattern would be skipped for smaller messages. This approach would result in a conditional claim-check pattern.

When to use this pattern

This pattern should be used whenever a message cannot fit the supported message limit of the chosen message bus technology. For example, Event Hubs currently has a limit of 256 KB (Basic Tier), while Event Grid supports only 64-KB messages.

The pattern can also be used if the payload should be accessed only by services that are authorized to see it. By offloading the payload to an external resource, stricter authentication and authorization rules can be put in place, to ensure that security is enforced when sensitive data is stored in the payload.

Examples

On Azure, this pattern can be implemented in several ways and with different technologies, but there are two main categories. In both cases, the receiver has the responsibility to read the claim check and use it to retrieve the payload.

- **Automatic claim-check generation**. This approach uses Azure Event Grid to automatically generate the claim check and push it into the message bus.
- **Manual claim-check generation**. In this approach, the sender is responsible for managing the payload. The sender stores the payload using the appropriate service, gets or generates the claim check, and sends the claim check to the message bus.

Event Grid is an event routing service and tries to deliver events within a configurable amount of time up to 24 hours. After that, events are either discarded or dead lettered. If you need to archive the event payloads or replay the event stream, you can add an Event Grid subscription to Event Hubs or Queue Storage, where messages can be retained for longer periods and archiving messages is supported. For information about fine tuning Event Grid message delivery and retry, and dead letter configuration, see Dead letter and retry policies.

Automatic claim-check generation with Blob Storage and Event Grid

In this approach, the sender drops the message payload into a designated Azure Blob Storage container. Event Grid automatically generates a tag/reference and sends it to a supported message bus, such as Azure Storage Queues. The receiver can poll the queue, get the message, and then use the stored reference data to download the payload directly from Blob Storage.

The same Event Grid message can be directly consumed by Azure Functions, without needing to go through a message bus. This approach takes full advantage of the serverless nature of both Event Grid and Functions.

You can find example code for this approach here.

Event Grid with Event Hubs

Similar to the previous example, Event Grid automatically generates a message when a payload is written to an Azure Blob container. But in this example, the message bus is implemented using Event Hubs. A client can register itself to receive the stream of messages as they are written to the event hub. The event hub can also be configured to archive messages, making them available as an Avro file that can be queried using tools like Apache Spark, Apache Drill, or any of the available Avro libraries.

You can find example code for this approach here.

Claim check generation with Service Bus

This solution takes advantage of a specific Service Bus plugin, ServiceBus.AttachmentPlugin, which makes the

claim-check workflow easy to implement. The plugin converts any message body into an attachment that gets stored in Azure Blob Storage when the message is sent.

```
using ServiceBus.AttachmentPlugin;
...

// Getting connection information
var serviceBusConnectionString = Environment.GetEnvironmentVariable("SERVICE_BUS_CONNECTION_STRING");
var queueName = Environment.GetEnvironmentVariable("QUEUE_NAME");
var storageConnectionString = Environment.GetEnvironmentVariable("STORAGE_CONNECTION_STRING");

// Creating config for sending message
var config = new AzureStorageAttachmentConfiguration(storageConnectionString);

// Creating and registering the sender using Service Bus Connection String and Queue Name
var sender = new MessageSender(serviceBusConnectionString, queueName);
sender.RegisterAzureStorageAttachmentPlugin(config);

// Create payload
var payload = new { data = "random data string for testing" };
var serialized = JsonConvert.SerializeObject(payload);
var payloadAsBytes = Encoding.UTF8.GetBytes(serialized);
var message = new Message(payloadAsBytes);

// Send the message
await sender.SendAsync(message);
```

The Service Bus message acts as a notification queue, which a client can subscribe to. When the consumer receives the message, the plugin makes it possible to directly read the message data from Blob Storage. You can then choose how to process the message further. An advantage of this approach is that it abstracts the claim-check workflow from the sender and receiver.

You can find example code for this approach here.

Manual claim-check generation with Kafka

In this example, a Kafka client writes the payload to Azure Blob Storage. Then it sends a notification message using Kafka-enabled Event Hubs. The consumer receives the message and can access the payload from Blob Storage. This example shows how a different messaging protocol can be used to implement the claim-check pattern in Azure. For example, you might need to support existing Kafka clients.

You can find example code for this approach here.

Next steps

- The examples described above are available on GitHub.
- The Enterprise Integration Patterns site has a description of this pattern.
- For another example, see Dealing with large Service Bus messages using claim check pattern (blog post).

Related guidance

- An alternative pattern for handling large messages is Split and Aggregate.

Compensating Transaction pattern

11/19/2021 • 7 minutes to read • Edit Online

Undo the work performed by a series of steps, which together define an eventually consistent operation, if one or more of the steps fail. Operations that follow the eventual consistency model are commonly found in cloud-hosted applications that implement complex business processes and workflows.

Context and problem

Applications running in the cloud frequently modify data. This data might be spread across various data sources held in different geographic locations. To avoid contention and improve performance in a distributed environment, an application shouldn't try to provide strong transactional consistency. Rather, the application should implement eventual consistency. In this model, a typical business operation consists of a series of separate steps. While these steps are being performed, the overall view of the system state might be inconsistent, but when the operation has completed and all of the steps have been executed the system should become consistent again.

> The Data Consistency Primer provides information about why distributed transactions don't scale well, and the principles of the eventual consistency model.

A challenge in the eventual consistency model is how to handle a step that has failed. In this case it might be necessary to undo all of the work completed by the previous steps in the operation. However, the data can't simply be rolled back because other concurrent instances of the application might have changed it. Even in cases where the data hasn't been changed by a concurrent instance, undoing a step might not simply be a matter of restoring the original state. It might be necessary to apply various business-specific rules (see the travel website described in the Example section).

If an operation that implements eventual consistency spans several heterogeneous data stores, undoing the steps in the operation will require visiting each data store in turn. The work performed in every data store must be undone reliably to prevent the system from remaining inconsistent.

Not all data affected by an operation that implements eventual consistency might be held in a database. In a service oriented architecture (SOA) environment an operation could invoke an action in a service, and cause a change in the state held by that service. To undo the operation, this state change must also be undone. This can involve invoking the service again and performing another action that reverses the effects of the first.

Solution

The solution is to implement a compensating transaction. The steps in a compensating transaction must undo the effects of the steps in the original operation. A compensating transaction might not be able to simply replace the current state with the state the system was in at the start of the operation because this approach could overwrite changes made by other concurrent instances of an application. Instead, it must be an intelligent process that takes into account any work done by concurrent instances. This process will usually be application specific, driven by the nature of the work performed by the original operation.

A common approach is to use a workflow to implement an eventually consistent operation that requires compensation. As the original operation proceeds, the system records information about each step and how the work performed by that step can be undone. If the operation fails at any point, the workflow rewinds back through the steps it's completed and performs the work that reverses each step. Note that a compensating transaction might not have to undo the work in the exact reverse order of the original operation, and it might be

possible to perform some of the undo steps in parallel.

> This approach is similar to the Sagas strategy discussed in Clemens Vasters' blog.

A compensating transaction is also an eventually consistent operation and it could also fail. The system should be able to resume the compensating transaction at the point of failure and continue. It might be necessary to repeat a step that's failed, so the steps in a compensating transaction should be defined as idempotent commands. For more information, see Idempotency Patterns on Jonathan Oliver's blog.

In some cases it might not be possible to recover from a step that has failed except through manual intervention. In these situations the system should raise an alert and provide as much information as possible about the reason for the failure.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

It might not be easy to determine when a step in an operation that implements eventual consistency has failed. A step might not fail immediately, but instead could block. It might be necessary to implement some form of time-out mechanism.

-Compensation logic isn't easily generalized. A compensating transaction is application specific. It relies on the application having sufficient information to be able to undo the effects of each step in a failed operation.

You should define the steps in a compensating transaction as idempotent commands. This enables the steps to be repeated if the compensating transaction itself fails.

The infrastructure that handles the steps in the original operation, and the compensating transaction, must be resilient. It must not lose the information required to compensate for a failing step, and it must be able to reliably monitor the progress of the compensation logic.

A compensating transaction doesn't necessarily return the data in the system to the state it was in at the start of the original operation. Instead, it compensates for the work performed by the steps that completed successfully before the operation failed.

The order of the steps in the compensating transaction doesn't necessarily have to be the exact opposite of the steps in the original operation. For example, one data store might be more sensitive to inconsistencies than another, and so the steps in the compensating transaction that undo the changes to this store should occur first.

Placing a short-term timeout-based lock on each resource that's required to complete an operation, and obtaining these resources in advance, can help increase the likelihood that the overall activity will succeed. The work should be performed only after all the resources have been acquired. All actions must be finalized before the locks expire.

Consider using retry logic that is more forgiving than usual to minimize failures that trigger a compensating transaction. If a step in an operation that implements eventual consistency fails, try handling the failure as a transient exception and repeat the step. Only stop the operation and initiate a compensating transaction if a step fails repeatedly or irrecoverably.

> Many of the challenges of implementing a compensating transaction are the same as those with implementing eventual consistency. See the section Considerations for Implementing Eventual Consistency in the Data Consistency Primer for more information.

When to use this pattern

Use this pattern only for operations that must be undone if they fail. If possible, design solutions to avoid the

complexity of requiring compensating transactions.

Example

A travel website lets customers book itineraries. A single itinerary might comprise a series of flights and hotels. A customer traveling from Seattle to London and then on to Paris could perform the following steps when creating an itinerary:

1. Book a seat on flight F1 from Seattle to London.
2. Book a seat on flight F2 from London to Paris.
3. Book a seat on flight F3 from Paris to Seattle.
4. Reserve a room at hotel H1 in London.
5. Reserve a room at hotel H2 in Paris.

These steps constitute an eventually consistent operation, although each step is a separate action. Therefore, as well as performing these steps, the system must also record the counter operations necessary to undo each step in case the customer decides to cancel the itinerary. The steps necessary to perform the counter operations can then run as a compensating transaction.

Notice that the steps in the compensating transaction might not be the exact opposite of the original steps, and the logic in each step in the compensating transaction must take into account any business-specific rules. For example, unbooking a seat on a flight might not entitle the customer to a complete refund of any money paid. The figure illustrates generating a compensating transaction to undo a long-running transaction to book a travel itinerary.

> **NOTE**
>
> It might be possible for the steps in the compensating transaction to be performed in parallel, depending on how you've designed the compensating logic for each step.

In many business solutions, failure of a single step doesn't always necessitate rolling the system back by using a compensating transaction. For example, if—after having booked flights F1, F2, and F3 in the travel website scenario—the customer is unable to reserve a room at hotel H1, it's preferable to offer the customer a room at a different hotel in the same city rather than canceling the flights. The customer can still decide to cancel (in which

case the compensating transaction runs and undoes the bookings made on flights F1, F2, and F3), but this decision should be made by the customer rather than by the system.

Related guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Data Consistency Primer. The Compensating Transaction pattern is often used to undo operations that implement the eventual consistency model. This primer provides information on the benefits and tradeoffs of eventual consistency.
- Scheduler-Agent-Supervisor pattern. Describes how to implement resilient systems that perform business operations that use distributed services and resources. Sometimes, it might be necessary to undo the work performed by an operation by using a compensating transaction.
- Retry pattern. Compensating transactions can be expensive to perform, and it might be possible to minimize their use by implementing an effective policy of retrying failing operations by following the Retry pattern.

Competing Consumers pattern

11/19/2021 • 9 minutes to read • Edit Online

Enable multiple concurrent consumers to process messages received on the same messaging channel. With multiple concurrent consumers, a system can process multiple messages concurrently to optimize throughput, to improve scalability and availability, and to balance the workload.

Context and problem

An application running in the cloud is expected to handle a large number of requests. Rather than process each request synchronously, a common technique is for the application to pass them through a messaging system to another service (a consumer service) that handles them asynchronously. This strategy helps to ensure that the business logic in the application isn't blocked, while the requests are being processed.

The number of requests can vary significantly over time for many reasons. A sudden increase in user activity or aggregated requests coming from multiple tenants can cause an unpredictable workload. At peak hours, a system might need to process many hundreds of requests per second, while at other times the number could be very small. Additionally, the nature of the work performed to handle these requests might be highly variable. By using a single instance of the consumer service, you can cause that instance to become flooded with requests. Or, the messaging system might be overloaded by an influx of messages that come from the application. To handle this fluctuating workload, the system can run multiple instances of the consumer service. However, these consumers must be coordinated to ensure that each message is only delivered to a single consumer. The workload also needs to be load balanced across consumers to prevent an instance from becoming a bottleneck.

Solution

Use a message queue to implement the communication channel between the application and the instances of the consumer service. The application posts requests in the form of messages to the queue, and the consumer service instances receive messages from the queue and process them. This approach enables the same pool of consumer service instances to handle messages from any instance of the application. The figure illustrates using a message queue to distribute work to instances of a service.

This solution has the following benefits:

- It provides a load-leveled system that can handle wide variations in the volume of requests sent by application instances. The queue acts as a buffer between the application instances and the consumer service instances. This buffer can help minimize the impact on availability and responsiveness, for both

the application and the service instances. For more information, see Queue-based Load Leveling pattern. Handling a message that requires some long-running processing doesn't prevent other messages from being handled concurrently by other instances of the consumer service.

- It improves reliability. If a producer communicates directly with a consumer instead of using this pattern, but doesn't monitor the consumer, there's a high probability that messages could be lost or fail to be processed if the consumer fails. In this pattern, messages aren't sent to a specific service instance. A failed service instance won't block a producer, and messages can be processed by any working service instance.
- It doesn't require complex coordination between the consumers, or between the producer and the consumer instances. The message queue ensures that each message is delivered at least once.
- It's scalable. The system can dynamically increase or decrease the number of instances of the consumer service as the volume of messages fluctuates.
- It can improve resiliency if the message queue provides transactional read operations. If a consumer service instance reads and processes the message as part of a transactional operation, and the consumer service instance fails, this pattern can ensure that the message will be returned to the queue to be picked up and handled by another instance of the consumer service.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- **Message ordering**. The order in which consumer service instances receive messages isn't guaranteed, and doesn't necessarily reflect the order in which the messages were created. Design the system to ensure that message processing is idempotent because this will help to eliminate any dependency on the order in which messages are handled. For more information, see Idempotency Patterns on Jonathon Oliver's blog.

  > Microsoft Azure Service Bus Queues can implement guaranteed first-in-first-out ordering of messages by using message sessions. For more information, see Messaging Patterns Using Sessions.

- **Designing services for resiliency**. If the system is designed to detect and restart failed service instances, it might be necessary to implement the processing performed by the service instances as idempotent operations to minimize the effects of a single message being retrieved and processed more than once.
- **Detecting poison messages**. A malformed message, or a task that requires access to resources that aren't available, can cause a service instance to fail. The system should prevent such messages being returned to the queue, and instead capture and store the details of these messages elsewhere so that they can be analyzed if necessary.
- **Handling results**. The service instance handling a message is fully decoupled from the application logic that generates the message, and they might not be able to communicate directly. If the service instance generates results that must be passed back to the application logic, this information must be stored in a location that's accessible to both. In order to prevent the application logic from retrieving incomplete data the system must indicate when processing is complete.

  > If you're using Azure, a worker process can pass results back to the application logic by using a dedicated message reply queue. The application logic must be able to correlate these results with the original message. This scenario is described in more detail in the Asynchronous Messaging Primer.

- **Scaling the messaging system**. In a large-scale solution, a single message queue could be overwhelmed by the number of messages and become a bottleneck in the system. In this situation,

consider partitioning the messaging system to send messages from specific producers to a particular queue, or use load balancing to distribute messages across multiple message queues.

- **Ensuring reliability of the messaging system**. A reliable messaging system is needed to guarantee that after the application enqueues a message it won't be lost. This system is essential for ensuring that all messages are delivered at least once.

When to use this pattern

Use this pattern when:

- The workload for an application is divided into tasks that can run asynchronously.
- Tasks are independent and can run in parallel.
- The volume of work is highly variable, requiring a scalable solution.
- The solution must provide high availability, and must be resilient if the processing for a task fails.

This pattern might not be useful when:

- It's not easy to separate the application workload into discrete tasks, or there's a high degree of dependence between tasks.
- Tasks must be performed synchronously, and the application logic must wait for a task to complete before continuing.
- Tasks must be performed in a specific sequence.

> Some messaging systems support sessions that enable a producer to group messages together and ensure that they're all handled by the same consumer. This mechanism can be used with prioritized messages (if they are supported) to implement a form of message ordering that delivers messages in sequence from a producer to a single consumer.

Example

Azure provides Service Bus Queues and Azure Function queue triggers that, when combined, are a direct implementation of this cloud design pattern. Azure Functions integrate with Azure Service Bus via triggers and bindings. Integrating with Service Bus allows you to build functions that consume queue messages sent by publishers. The publishing application(s) will post messages to a queue, and consumers, implemented as Azure Functions, can retrieve messages from this queue and handle them.

For resiliency, a Service Bus queue enables a consumer to use `PeekLock` mode when it retrieves a message from the queue; this mode doesn't actually remove the message, but simply hides it from other consumers. The Azure Functions runtime receives a message in PeekLock mode, if the function finishes successfully it calls Complete on the message, or it may call Abandon if the function fails, and the message will become visible again, allowing another consumer to retrieve it. If the function runs for a period longer than the PeekLock timeout, the lock is automatically renewed as long as the function is running.

Azure Functions can scale out/in based on the depth of the queue, all acting as competing consumers of the queue. If multiple instances of the functions are created they all compete by independently pulling and processing the messages.

For detailed information on using Azure Service Bus queues, see Service Bus queues, topics, and subscriptions.

For information on Queue triggered Azure Functions, see Azure Service Bus trigger for Azure Functions.

The following code shows how you can create a new message and send it to a Service Bus Queue by using a `ServiceBusClient` instance.

```
private string serviceBusConnectionString = ...;
...

  public async Task SendMessagesAsync(CancellationToken  ct)
  {
   try
   {
    var msgNumber = 0;

    var serviceBusClient = new ServiceBusClient(serviceBusConnectionString);

    // create the sender
    ServiceBusSender sender = serviceBusClient.CreateSender("myqueue");

    while (!ct.IsCancellationRequested)
    {
     // Create a new message to send to the queue
     string messageBody = $"Message {msgNumber}";
     var message = new ServiceBusMessage(messageBody);

     // Write the body of the message to the console
     this._logger.LogInformation($"Sending message: {messageBody}");

     // Send the message to the queue
     await sender.SendMessageAsync(message);

     this._logger.LogInformation("Message successfully sent.");
     msgNumber++;
    }
   }
   catch (Exception exception)
   {
    this._logger.LogException(exception.Message);
   }
  }
```

The following code example shows a consumer, written as a C# Azure Function, that reads message metadata and logs a Service Bus Queue message. Note how the `ServiceBusTrigger` attribute is used to bind it to a Service Bus Queue.

```
[FunctionName("ProcessQueueMessage")]
public static void Run(
    [ServiceBusTrigger("myqueue", Connection = "ServiceBusConnectionString")]
    string myQueueItem,
    Int32 deliveryCount,
    DateTime enqueuedTimeUtc,
    string messageId,
    ILogger log)
{
    log.LogInformation($"C# ServiceBus queue trigger function consumed message: {myQueueItem}");
    log.LogInformation($"EnqueuedTimeUtc={enqueuedTimeUtc}");
    log.LogInformation($"DeliveryCount={deliveryCount}");
    log.LogInformation($"MessageId={messageId}");
}
```

Related guidance

The following patterns and guidance might be relevant when implementing this pattern:

- Asynchronous Messaging Primer. Message queues are an asynchronous communications mechanism. If a consumer service needs to send a reply to an application, it might be necessary to implement some form of response messaging. The Asynchronous Messaging Primer provides information on how to

implement request/reply messaging using message queues.

- Autoscaling Guidance. It might be possible to start and stop instances of a consumer service since the length of the queue applications post messages on varies. Autoscaling can help to maintain throughput during times of peak processing.
- Compute Resource Consolidation pattern. It might be possible to consolidate multiple instances of a consumer service into a single process to reduce costs and management overhead. The Compute Resource Consolidation pattern describes the benefits and tradeoffs of following this approach.
- Queue-based Load Leveling pattern. Introducing a message queue can add resiliency to the system, enabling service instances to handle widely varying volumes of requests from application instances. The message queue acts as a buffer, which levels the load. The Queue-based Load Leveling pattern describes this scenario in more detail.

Compute Resource Consolidation pattern

11/19/2021 • 12 minutes to read • Edit Online

Consolidate multiple tasks or operations into a single computational unit. This can increase compute resource utilization, and reduce the costs and management overhead associated with performing compute processing in cloud-hosted applications.

Context and problem

A cloud application often implements a variety of operations. In some solutions it makes sense to follow the design principle of separation of concerns initially, and divide these operations into separate computational units that are hosted and deployed individually (for example, as separate App Service web apps, separate Virtual Machines, or separate Cloud Service roles). However, although this strategy can help simplify the logical design of the solution, deploying a large number of computational units as part of the same application can increase runtime hosting costs and make management of the system more complex.

As an example, the figure shows the simplified structure of a cloud-hosted solution that is implemented using more than one computational unit. Each computational unit runs in its own virtual environment. Each function has been implemented as a separate task (labeled Task A through Task E) running in its own computational unit.

Each computational unit consumes chargeable resources, even when it's idle or lightly used. Therefore, this isn't always the most cost-effective solution.

In Azure, this concern applies to roles in a Cloud Service, App Services, and Virtual Machines. These items run in their own virtual environment. Running a collection of separate roles, websites, or virtual machines that are designed to perform a set of well-defined operations, but that need to communicate and cooperate as part of a single solution, can be an inefficient use of resources.

Solution

To help reduce costs, increase utilization, improve communication speed, and reduce management it's possible to consolidate multiple tasks or operations into a single computational unit.

Tasks can be grouped according to criteria based on the features provided by the environment and the costs associated with these features. A common approach is to look for tasks that have a similar profile concerning

their scalability, lifetime, and processing requirements. Grouping these together allows them to scale as a unit. The elasticity provided by many cloud environments enables additional instances of a computational unit to be started and stopped according to the workload. For example, Azure provides autoscaling that you can apply to roles in a Cloud Service, App Services, and Virtual Machines. For more information, see Autoscaling Guidance.

As a counter example to show how scalability can be used to determine which operations shouldn't be grouped together, consider the following two tasks:

- Task 1 polls for infrequent, time-insensitive messages sent to a queue.
- Task 2 handles high-volume bursts of network traffic.

The second task requires elasticity that can involve starting and stopping a large number of instances of the computational unit. Applying the same scaling to the first task would simply result in more tasks listening for infrequent messages on the same queue, and is a waste of resources.

In many cloud environments it's possible to specify the resources available to a computational unit in terms of the number of CPU cores, memory, disk space, and so on. Generally, the more resources specified, the greater the cost. To save money, it's important to maximize the work an expensive computational unit performs, and not let it become inactive for an extended period.

If there are tasks that require a great deal of CPU power in short bursts, consider consolidating these into a single computational unit that provides the necessary power. However, it's important to balance this need to keep expensive resources busy against the contention that could occur if they are over stressed. Long-running, compute-intensive tasks shouldn't share the same computational unit, for example.

Issues and considerations

Consider the following points when implementing this pattern:

Scalability and elasticity. Many cloud solutions implement scalability and elasticity at the level of the computational unit by starting and stopping instances of units. Avoid grouping tasks that have conflicting scalability requirements in the same computational unit.

Lifetime. The cloud infrastructure periodically recycles the virtual environment that hosts a computational unit. When there are many long-running tasks inside a computational unit, it might be necessary to configure the unit to prevent it from being recycled until these tasks have finished. Alternatively, design the tasks by using a check-pointing approach that enables them to stop cleanly, and continue at the point they were interrupted when the computational unit is restarted.

Release cadence. If the implementation or configuration of a task changes frequently, it might be necessary to stop the computational unit hosting the updated code, reconfigure and redeploy the unit, and then restart it. This process will also require that all other tasks within the same computational unit are stopped, redeployed, and restarted.

Security. Tasks in the same computational unit might share the same security context and be able to access the same resources. There must be a high degree of trust between the tasks, and confidence that one task isn't going to corrupt or adversely affect another. Additionally, increasing the number of tasks running in a computational unit increases the attack surface of the unit. Each task is only as secure as the one with the most vulnerabilities.

Fault tolerance. If one task in a computational unit fails or behaves abnormally, it can affect the other tasks running within the same unit. For example, if one task fails to start correctly it can cause the entire startup logic for the computational unit to fail, and prevent other tasks in the same unit from running.

Contention. Avoid introducing contention between tasks that compete for resources in the same computational unit. Ideally, tasks that share the same computational unit should exhibit different resource utilization characteristics. For example, two compute-intensive tasks should probably not reside in the same

computational unit, and neither should two tasks that consume large amounts of memory. However, mixing a compute-intensive task with a task that requires a large amount of memory is a workable combination.

> **NOTE**
> Consider consolidating compute resources only for a system that's been in production for a period of time so that operators and developers can monitor the system and create a *heat map* that identifies how each task uses differing resources. This map can be used to determine which tasks are good candidates for sharing compute resources.

Complexity. Combining multiple tasks into a single computational unit adds complexity to the code in the unit, possibly making it more difficult to test, debug, and maintain.

Stable logical architecture. Design and implement the code in each task so that it shouldn't need to change, even if the physical environment the task runs in does change.

Other strategies. Consolidating compute resources is only one way to help reduce costs associated with running multiple tasks concurrently. It requires careful planning and monitoring to ensure that it remains an effective approach. Other strategies might be more appropriate, depending on the nature of the work and where the users these tasks are running are located. For example, functional decomposition of the workload (as described by the Compute Partitioning Guidance) might be a better option.

When to use this pattern

Use this pattern for tasks that are not cost effective if they run in their own computational units. If a task spends much of its time idle, running this task in a dedicated unit can be expensive.

This pattern might not be suitable for tasks that perform critical fault-tolerant operations, or tasks that process highly sensitive or private data and require their own security context. These tasks should run in their own isolated environment, in a separate computational unit.

Example

When building a cloud service on Azure, it's possible to consolidate the processing performed by multiple tasks into a single role. Typically this is a worker role that performs background or asynchronous processing tasks.

> In some cases it's possible to include background or asynchronous processing tasks in the web role. This technique helps to reduce costs and simplify deployment, although it can impact the scalability and responsiveness of the public-facing interface provided by the web role.

The role is responsible for starting and stopping the tasks. When the Azure fabric controller loads a role, it raises the `Start` event for the role. You can override the `OnStart` method of the `WebRole` or `WorkerRole` class to handle this event, perhaps to initialize the data and other resources the tasks in this method depend on.

When the `OnStart` method completes, the role can start responding to requests. You can find more information and guidance about using the `OnStart` and `Run` methods in a role in the Application Startup Processes section in the patterns & practices guide Moving Applications to the Cloud.

> Keep the code in the `OnStart` method as concise as possible. Azure doesn't impose any limit on the time taken for this method to complete, but the role won't be able to start responding to network requests sent to it until this method completes.

When the `OnStart` method has finished, the role executes the `Run` method. At this point, the fabric controller can start sending requests to the role.

Place the code that actually creates the tasks in the `Run` method. Note that the `Run` method defines the lifetime of the role instance. When this method completes, the fabric controller will arrange for the role to be shut down.

When a role shuts down or is recycled, the fabric controller prevents any more incoming requests being received from the load balancer and raises the `Stop` event. You can capture this event by overriding the `OnStop` method of the role and perform any tidying up required before the role terminates.

> Any actions performed in the `OnStop` method must be completed within five minutes (or 30 seconds if you are using the Azure emulator on a local computer). Otherwise the Azure fabric controller assumes that the role has stalled and will force it to stop.

The tasks are started by the `Run` method that waits for the tasks to complete. The tasks implement the business logic of the cloud service, and can respond to messages posted to the role through the Azure load balancer. The figure shows the lifecycle of tasks and resources in a role in an Azure cloud service.

The *WorkerRole.cs* file in the *ComputeResourceConsolidation.Worker* project shows an example of how you might implement this pattern in an Azure cloud service.

> The *ComputeResourceConsolidation.Worker* project is part of the *ComputeResourceConsolidation* solution available for download from GitHub.

The `MyWorkerTask1` and the `MyWorkerTask2` methods illustrate how to perform different tasks within the same worker role. The following code shows `MyWorkerTask1`. This is a simple task that sleeps for 30 seconds and then outputs a trace message. It repeats this process until the task is canceled. The code in `MyWorkerTask2` is similar.

```
// A sample worker role task.
private static async Task MyWorkerTask1(CancellationToken ct)
{
  // Fixed interval to wake up and check for work and/or do work.
  var interval = TimeSpan.FromSeconds(30);

  try
  {
    while (!ct.IsCancellationRequested)
    {
      // Wake up and do some background processing if not canceled.
      // TASK PROCESSING CODE HERE
      Trace.TraceInformation("Doing Worker Task 1 Work");

      // Go back to sleep for a period of time unless asked to cancel.
      // Task.Delay will throw an OperationCanceledException when canceled.
      await Task.Delay(interval, ct);
    }
  }
  catch (OperationCanceledException)
  {
    // Expect this exception to be thrown in normal circumstances or check
    // the cancellation token. If the role instances are shutting down, a
    // cancellation request will be signaled.
    Trace.TraceInformation("Stopping service, cancellation requested");

    // Rethrow the exception.
    throw;
  }
}
```

> The sample code shows a common implementation of a background process. In a real world application you can follow this same structure, except that you should place your own processing logic in the body of the loop that waits for the cancellation request.

After the worker role has initialized the resources it uses, the `Run` method starts the two tasks concurrently, as shown here.

```
/// <summary>
/// The cancellation token source use to cooperatively cancel running tasks
/// </summary>
private readonly CancellationTokenSource cts = new CancellationTokenSource();

/// <summary>
/// List of running tasks on the role instance
/// </summary>
private readonly List<Task> tasks = new List<Task>();

// RoleEntry Run() is called after OnStart().
// Returning from Run() will cause a role instance to recycle.
public override void Run()
{
  // Start worker tasks and add to the task list
  tasks.Add(MyWorkerTask1(cts.Token));
  tasks.Add(MyWorkerTask2(cts.Token));

  foreach (var worker in this.workerTasks)
  {
      this.tasks.Add(worker);
  }

  Trace.TraceInformation("Worker host tasks started");
  // The assumption is that all tasks should remain running and not return,
  // similar to role entry Run() behavior.
  try
  {
    Task.WaitAll(tasks.ToArray());
  }
  catch (AggregateException ex)
  {
    Trace.TraceError(ex.Message);

    // If any of the inner exceptions in the aggregate exception
    // are not cancellation exceptions then re-throw the exception.
    ex.Handle(innerEx => (innerEx is OperationCanceledException));
  }

  // If there wasn't a cancellation request, stop all tasks and return from Run()
  // An alternative to canceling and returning when a task exits would be to
  // restart the task.
  if (!cts.IsCancellationRequested)
  {
    Trace.TraceInformation("Task returned without cancellation request");
    Stop(TimeSpan.FromMinutes(5));
  }
}
...
```

In this example, the `Run` method waits for tasks to be completed. If a task is canceled, the `Run` method assumes that the role is being shut down and waits for the remaining tasks to be canceled before finishing (it waits for a maximum of five minutes before terminating). If a task fails due to an expected exception, the `Run` method cancels the task.

> You could implement more comprehensive monitoring and exception handling strategies in the `Run` method such as restarting tasks that have failed, or including code that enables the role to stop and start individual tasks.

The `Stop` method shown in the following code is called when the fabric controller shuts down the role instance (it's invoked from the `OnStop` method). The code stops each task gracefully by canceling it. If any task takes more than five minutes to complete, the cancellation processing in the `Stop` method ceases waiting and the

role is terminated.

```
// Stop running tasks and wait for tasks to complete before returning
// unless the timeout expires.
private void Stop(TimeSpan timeout)
{
  Trace.TraceInformation("Stop called. Canceling tasks.");
  // Cancel running tasks.
  cts.Cancel();

  Trace.TraceInformation("Waiting for canceled tasks to finish and return");

  // Wait for all the tasks to complete before returning. Note that the
  // emulator currently allows 30 seconds and Azure allows five
  // minutes for processing to complete.
  try
  {
    Task.WaitAll(tasks.ToArray(), timeout);
  }
  catch (AggregateException ex)
  {
    Trace.TraceError(ex.Message);

    // If any of the inner exceptions in the aggregate exception
    // are not cancellation exceptions then rethrow the exception.
    ex.Handle(innerEx => (innerEx is OperationCanceledException));
  }
}
```

Related guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Autoscaling Guidance. Autoscaling can be used to start and stop instances of service hosting computational resources, depending on the anticipated demand for processing.
- Compute Partitioning Guidance. Describes how to allocate the services and components in a cloud service in a way that helps to minimize running costs while maintaining the scalability, performance, availability, and security of the service.
- This pattern includes a downloadable sample application.

CQRS pattern

11/19/2021 • 11 minutes to read • Edit Online

CQRS stands for Command and Query Responsibility Segregation, a pattern that separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security. The flexibility created by migrating to CQRS allows a system to better evolve over time and prevents update commands from causing merge conflicts at the domain level.

Context and problem

In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations. In more complex applications, however, this approach can become unwieldy. For example, on the read side, the application may perform many different queries, returning data transfer objects (DTOs) with different shapes. Object mapping can become complicated. On the write side, the model may implement complex validation and business logic. As a result, you can end up with an overly complex model that does too much.

Read and write workloads are often asymmetrical, with very different performance and scale requirements.

- There is often a mismatch between the read and write representations of the data, such as additional columns or properties that must be updated correctly even though they aren't required as part of an operation.
- Data contention can occur when operations are performed in parallel on the same set of data.
- The traditional approach can have a negative effect on performance due to load on the data store and data access layer, and the complexity of queries required to retrieve information.
- Managing security and permissions can become complex, because each entity is subject to both read and write operations, which might expose data in the wrong context.

Solution

CQRS separates reads and writes into different models, using **commands** to update data, and **queries** to read data.

- Commands should be task-based, rather than data centric. ("Book hotel room", not "set ReservationStatus to Reserved").
- Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

The models can then be isolated, as shown in the following diagram, although that's not an absolute

requirement.

Having separate query and update models simplifies the design and implementation. However, one disadvantage is that CQRS code can't automatically be generated from a database schema using scaffolding mechanisms such as O/RM tools.

For greater isolation, you can physically separate the read data from the write data. In that case, the read database can use its own data schema that is optimized for queries. For example, it can store a materialized view of the data, in order to avoid complex joins or complex O/RM mappings. It might even use a different type of data store. For example, the write database might be relational, while the read database is a document database.

If separate read and write databases are used, they must be kept in sync. Typically this is accomplished by having the write model publish an event whenever it updates the database. For more information about using events, see Event-driven architecture style. Updating the database and publishing the event must occur in a single transaction.

The read store can be a read-only replica of the write store, or the read and write stores can have a different structure altogether. Using multiple read-only replicas can increase query performance, especially in distributed scenarios where read-only replicas are located close to the application instances.

Separation of the read and write stores also allows each to be scaled appropriately to match the load. For example, read stores typically encounter a much higher load than write stores.

Some implementations of CQRS use the Event Sourcing pattern. With this pattern, application state is stored as a sequence of events. Each event represents a set of changes to the data. The current state is constructed by replaying the events. In a CQRS context, one benefit of Event Sourcing is that the same events can be used to notify other components — in particular, to notify the read model. The read model uses the events to create a snapshot of the current state, which is more efficient for queries. However, Event Sourcing adds complexity to the design.

Benefits of CQRS include:

- **Independent scaling**. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.

- **Optimized data schemas**. The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
- **Security**. It's easier to ensure that only the right domain entities are performing writes on the data.
- **Separation of concerns**. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
- **Simpler queries**. By storing a materialized view in the read database, the application can avoid complex joins when querying.

Implementation issues and considerations

Some challenges of implementing this pattern include:

- **Complexity**. The basic idea of CQRS is simple. But it can lead to a more complex application design, especially if they include the Event Sourcing pattern.

- **Messaging**. Although CQRS does not require messaging, it's common to use messaging to process commands and publish update events. In that case, the application must handle message failures or duplicate messages. See the guidance on Priority Queues for dealing with commands having different priorities.

- **Eventual consistency**. If you separate the read and write databases, the read data may be stale. The read model store must be updated to reflect changes to the write model store, and it can be difficult to detect when a user has issued a request based on stale read data.

When to use CQRS pattern

Consider CQRS for the following scenarios:

- Collaborative domains where many users access the same data in parallel. CQRS allows you to define commands with enough granularity to minimize merge conflicts at the domain level, and conflicts that do arise can be merged by the command.

- Task-based user interfaces where users are guided through a complex process as a series of steps or with complex domain models. The write model has a full command-processing stack with business logic, input validation, and business validation. The write model may treat a set of associated objects as a single unit for data changes (an aggregate, in DDD terminology) and ensure that these objects are always in a consistent state. The read model has no business logic or validation stack, and just returns a DTO for use in a view model. The read model is eventually consistent with the write model.

- Scenarios where performance of data reads must be fine-tuned separately from performance of data writes, especially when the number of reads is much greater than the number of writes. In this scenario, you can scale out the read model, but run the write model on just a few instances. A small number of write model instances also helps to minimize the occurrence of merge conflicts.

- Scenarios where one team of developers can focus on the complex domain model that is part of the write model, and another team can focus on the read model and the user interfaces.

- Scenarios where the system is expected to evolve over time and might contain multiple versions of the model, or where business rules change regularly.

- Integration with other systems, especially in combination with event sourcing, where the temporal failure of one subsystem shouldn't affect the availability of the others.

This pattern isn't recommended when:

- The domain or the business rules are simple.

- A simple CRUD-style user interface and data access operations are sufficient.

Consider applying CQRS to limited sections of your system where it will be most valuable.

Event Sourcing and CQRS pattern

The CQRS pattern is often used along with the Event Sourcing pattern. CQRS-based systems use separate read and write data models, each tailored to relevant tasks and often located in physically separate stores. When used with the Event Sourcing pattern, the store of events is the write model, and is the official source of information. The read model of a CQRS-based system provides materialized views of the data, typically as highly denormalized views. These views are tailored to the interfaces and display requirements of the application, which helps to maximize both display and query performance.

Using the stream of events as the write store, rather than the actual data at a point in time, avoids update conflicts on a single aggregate and maximizes performance and scalability. The events can be used to asynchronously generate materialized views of the data that are used to populate the read store.

Because the event store is the official source of information, it is possible to delete the materialized views and replay all past events to create a new representation of the current state when the system evolves, or when the read model must change. The materialized views are in effect a durable read-only cache of the data.

When using CQRS combined with the Event Sourcing pattern, consider the following:

- As with any system where the write and read stores are separate, systems based on this pattern are only eventually consistent. There will be some delay between the event being generated and the data store being updated.
- The pattern adds complexity because code must be created to initiate and handle events, and assemble or update the appropriate views or objects required by queries or a read model. The complexity of the CQRS pattern when used with the Event Sourcing pattern can make a successful implementation more difficult, and requires a different approach to designing systems. However, event sourcing can make it easier to model the domain, and makes it easier to rebuild views or create new ones because the intent of the changes in the data is preserved.
- Generating materialized views for use in the read model or projections of the data by replaying and handling the events for specific entities or collections of entities can require significant processing time and resource usage. This is especially true if it requires summation or analysis of values over long periods, because all the associated events might need to be examined. Resolve this by implementing snapshots of the data at scheduled intervals, such as a total count of the number of a specific action that has occurred, or the current state of an entity.

Example of CQRS pattern

The following code shows some extracts from an example of a CQRS implementation that uses different definitions for the read and the write models. The model interfaces don't dictate any features of the underlying data stores, and they can evolve and be fine-tuned independently because these interfaces are separated.

The following code shows the read model definition.

```
// Query interface
namespace ReadModel
{
  public interface ProductsDao
  {
    ProductDisplay FindById(int productId);
    ICollection<ProductDisplay> FindByName(string name);
    ICollection<ProductInventory> FindOutOfStockProducts();
    ICollection<ProductDisplay> FindRelatedProducts(int productId);
  }

  public class ProductDisplay
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal UnitPrice { get; set; }
    public bool IsOutOfStock { get; set; }
    public double UserRating { get; set; }
  }

  public class ProductInventory
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public int CurrentStock { get; set; }
  }
}
```

The system allows users to rate products. The application code does this using the `RateProduct` command shown in the following code.

```
public interface ICommand
{
  Guid Id { get; }
}

public class RateProduct : ICommand
{
  public RateProduct()
  {
    this.Id = Guid.NewGuid();
  }
  public Guid Id { get; set; }
  public int ProductId { get; set; }
  public int Rating { get; set; }
  public int UserId {get; set; }
}
```

The system uses the `ProductsCommandHandler` class to handle commands sent by the application. Clients typically send commands to the domain through a messaging system such as a queue. The command handler accepts these commands and invokes methods of the domain interface. The granularity of each command is designed to reduce the chance of conflicting requests. The following code shows an outline of the `ProductsCommandHandler` class.

```
public class ProductsCommandHandler :
    ICommandHandler<AddNewProduct>,
    ICommandHandler<RateProduct>,
    ICommandHandler<AddToInventory>,
    ICommandHandler<ConfirmItemShipped>,
    ICommandHandler<UpdateStockFromInventoryRecount>
{
  private readonly IRepository<Product> repository;

  public ProductsCommandHandler (IRepository<Product> repository)
  {
    this.repository = repository;
  }

  void Handle (AddNewProduct command)
  {
    ...
  }

  void Handle (RateProduct command)
  {
    var product = repository.Find(command.ProductId);
    if (product != null)
    {
      product.RateProduct(command.UserId, command.Rating);
      repository.Save(product);
    }
  }

  void Handle (AddToInventory command)
  {
    ...
  }

  void Handle (ConfirmItemsShipped command)
  {
    ...
  }

  void Handle (UpdateStockFromInventoryRecount command)
  {
    ...
  }
}
```

Next steps

The following patterns and guidance are useful when implementing this pattern:

- Data Consistency Primer. Explains the issues that are typically encountered due to eventual consistency between the read and write data stores when using the CQRS pattern, and how these issues can be resolved.
- Horizontal, vertical, and functional data partitioning. Describes best practices for dividing data into partitions that can be managed and accessed separately to improve scalability, reduce contention, and optimize performance.
- The patterns & practices guide CQRS Journey. In particular, Introducing the Command Query Responsibility Segregation pattern explores the pattern and when it's useful, and Epilogue: Lessons Learned helps you understand some of the issues that come up when using this pattern.

Martin Fowler's blog posts:

- What do you mean by "Event-Driven"?
- CQRS

Related guidance

- Event Sourcing pattern. Describes in more detail how Event Sourcing can be used with the CQRS pattern to simplify tasks in complex domains while improving performance, scalability, and responsiveness. As well as how to provide consistency for transactional data while maintaining full audit trails and history that can enable compensating actions.
- Materialized View pattern. The read model of a CQRS implementation can contain materialized views of the write model data, or the read model can be used to generate materialized views.

Deployment Stamps pattern

11/19/2021 • 12 minutes to read • Edit Online

The deployment stamp pattern involves provisioning, managing, and monitoring a heterogeneous group of resources to host and operate multiple workloads or tenants. Each individual copy is called a *stamp*, or sometimes a *service unit* or *scale unit*. In a multi-tenant environment, every stamp or scale unit can serve a predefined number of tenants. Multiple stamps can be deployed to scale the solution almost linearly and serve an increasing number of tenants. This approach can improve the scalability of your solution, allow you to deploy instances across multiple regions, and separate your customer data.

Context and problem

When hosting an application in the cloud there are certain considerations to be made. One key thing to keep in mind is the performance and reliability of your application. If you host a single instance of your solution, you might be subject to the following limitations:

- **Scale limits.** Deploying a single instance of your application may result in natural scaling limits. For example, you may use services that have limits on the number of inbound connections, host names, TCP sockets, or other resources.
- **Non-linear scaling or cost.** Some of your solution's components may not scale linearly with the number of requests or the amount of data. Instead, there can be a sudden decrease in performance or increase in cost once a threshold has been met. For example, you may use a database and discover that the marginal cost of adding more capacity (scaling up) becomes prohibitive, and that scaling out is a more cost-effective strategy. Similarly, Azure Front Door has higher per-domain pricing when a high number of custom domains are deployed, and it may be better to spread the custom domains across multiple Front Door instances.
- **Separation of customers.** You may need to keep certain customers' data isolated from other customers' data. Similarly, you may have some customers that require more system resources to service than others, and consider grouping them on different sets of infrastructure.
- **Handling single- and multi-tenant instances.** You may have some large customers who need their own independent instances of your solution. You may also have a pool of smaller customers who can share a multi-tenant deployment.
- **Complex deployment requirements.** You may need to deploy updates to your service in a controlled manner, and to deploy to different subsets of your customer base at different times.
- **Update frequency.** You may have some customers who are tolerant of having frequent updates to your system, while others may be risk-averse and want infrequent updates to the system that services their requests. It may make sense to have these customers deployed to isolated environments.
- **Geographical or geopolitical restrictions.** To architect for low latency, or to comply with data sovereignty requirements, you may deploy some of your customers into specific regions.

Solution

To avoid these issues, consider grouping resources in *scale units* and provisioning multiple copies of your *stamps*. Each *scale unit* will host and serve a subset of your tenants. Stamps operate independently of each other and can be deployed and updated independently. A single geographical region may contain a single stamp, or may contain multiple stamps to allow for horizontal scale-out within the region. Stamps contain a subset of your customers.

Deployment stamps can apply whether your solution uses infrastructure as a service (IaaS) or platform as a service (PaaS) components, or a mixture of both. Typically IaaS workloads require more intervention to scale, so the pattern may be useful for IaaS-heavy workloads to allow for scaling out.

Stamps can be used to implement deployment rings. If different customers want to receive service updates at different frequencies, they can be grouped onto different stamps, and each stamp could have updates deployed at different cadences.

Because stamps run independently from each other, data is implicitly *sharded*. Furthermore, a single stamp can make use of further sharding to internally allow for scalability and elasticity within the stamp.

The deployment stamp pattern is used internally by many Azure services, including App Service, Azure Stack, and Azure Storage.

Deployment stamps are related to, but distinct from, geodes. In a deployment stamp architecture, multiple independent instances of your system are deployed and contain a subset of your customers and users. In geodes, all instances can serve requests from any users, but this architecture is often more complex to design and build. You may also consider mixing the two patterns within one solution; the traffic routing approach described below is an example of such a hybrid scenario.

Deployment

Because of the complexity that is involved in deploying identical copies of the same components, good DevOps practices are critical to ensure success when implementing this pattern. Consider describing your infrastructure as code, such as by using Bicep, JSON Azure Resource Manager templates (ARM templates), Terraform, and scripts. With this approach, you can ensure that the deployment of each stamp is predictable and repeatable. It also reduces the likelihood of human errors such as accidental mismatches in configuration between stamps.

You can deploy updates automatically to all stamps in parallel, in which case you might consider technologies like Resource Manager templates or Azure Deployment Manager to coordinate the deployment of your infrastructure and applications. Alternatively, you may decide to gradually roll out updates to some stamps first, and then progressively to others. Azure Deployment Manager can also manage this type of staged rollout, or you could consider using a release management tool like Azure Pipelines to orchestrate deployments to each stamp. For more information, see:

- Integrate Bicep with Azure Pipelines
- Integrate JSON ARM templates with Azure Pipelines

Carefully consider the topology of the Azure subscriptions and resource groups for your deployments:

- Typically a subscription will contain all resources for a single solution, so in general consider using a single subscription for all stamps. However, some Azure services impose subscription-wide quotas, so if you are using this pattern to allow for a high degree of scale-out, you may need to consider deploying stamps across different subscriptions.
- Resource groups are generally used to deploy components with the same lifecycle. If you plan to deploy updates to all of your stamps at once, consider using a single resource group to contain all of the components for all of your stamps, and use resource naming conventions and tags to identify the components that belong to each stamp. Alternatively, if you plan to deploy updates to each stamp independently, consider deploying each stamp into its own resource group.

Capacity planning

Use load and performance testing to determine the approximate load that a given stamp can accommodate. Load metrics may be based on the number of customers/tenants that a single stamp can accommodate, or metrics from the services that the components within the stamp emit. Ensure that you have sufficient instrumentation to measure when a given stamp is approaching its capacity, and the ability to deploy new stamps quickly to respond to demand.

Traffic routing

The Deployment Stamp pattern works well if each stamp is addressed independently. For example, if Contoso deploys the same API application across multiple stamps, they might consider using DNS to route traffic to the relevant stamp:

- `unit1.aus.myapi.contoso.com` routes traffic to stamp `unit1` within an Australian region.
- `unit2.aus.myapi.contoso.com` routes traffic to stamp `unit2` within an Australian region.
- `unit1.eu.myapi.contoso.com` routes traffic to stamp `unit1` within a European region.

Clients are then responsible for connecting to the correct stamp.

If a single ingress point for all traffic is required, a traffic routing service may be used to resolve the stamp for a given request, customer, or tenant. The traffic routing service either directs the client to the relevant URL for the stamp (for example, using an HTTP 302 response status code), or may act as a reverse proxy and forward the traffic to the relevant stamp without the client being aware.

A centralized traffic routing service can be a complex component to design, especially when a solution runs across multiple regions. Consider deploying the traffic routing service into multiple regions (potentially including every region that stamps are deployed into), and then ensuring the data store (mapping tenants to stamps) is synchronized. The traffic routing component may itself by an instance of the geode pattern.

For example, Azure API Management could be deployed to act in the traffic routing service role. It can determine the appropriate stamp for a request by looking up data in a Cosmos DB collection storing the mapping between tenants and stamps. API Management can then dynamically set the back-end URL to the relevant stamp's API service.

To enable geo-distribution of requests and geo-redundancy of the traffic routing service, API Management can be deployed across multiple regions, or Azure Front Door can be used to direct traffic to the closest instance. Front Door can be configured with a backend pool, enabling requests to be directed to the closest available API Management instance. If you application is not exposed via HTTP/S, you can use a cross-region Azure Load Balancer to distribute incoming calls to regional Azure Load Balancers. The global distribution feature of Azure Cosmos DB can be used to keep the mapping information updated across each region.

If a traffic routing service is included in your solution, consider whether it acts as a gateway and could therefore perform gateway offloading for services such as token validation, throttling, and authorization.

Issues and considerations

You should consider the following points when deciding how to implement this pattern:

- **Deployment process.** When deploying multiple stamps, it is highly advisable to have automated and fully repeatable deployment processes. Consider using Bicep, JSON ARM templates, or Terraform modules to declaratively define the stamp.
- **Cross-stamp operations.** When your solution is deployed independently across multiple stamps, questions like "how many customers do we have across all of our stamps?" can become more complex to answer. Queries may need to be executed against each stamp and the results aggregated. Alternatively, consider having all of the stamps publish data into a centralized data warehouse for consolidated reporting.
- **Determining scale-out policies.** Stamps have a finite capacity, which might be defined using a proxy metric such as the number of tenants that can be deployed to the stamp. It is important to monitor the available capacity and used capacity for each stamp, and to proactively deploy additional stamps to allow for new tenants to be directed to them.
- **Cost.** The Deployment Stamp pattern involves deploying multiple copies of your infrastructure component, which will likely involve a substantial increase in the cost of operating your solution.
- **Moving between stamps.** As each stamp is deployed and operated independently, moving tenants between stamps can be difficult. Your application would need custom logic to transmit the information about a given customer to a different stamp, and then to remove the tenant's information from the original stamp. This process may require a backplane for communication between stamps, further increasing the complexity of the overall solution.
- **Traffic routing.** As described above, routing traffic to the correct stamp for a given request can require an additional component to resolve tenants to stamps. This component, in turn, may need to be made highly available.
- **Shared components.** You may have some components that can be shared across stamps. For example, if you have a shared single-page app for all tenants, consider deploying that into one region and using Azure CDN to replicate it globally.

When to use this pattern

This pattern is useful when you have:

- Natural limits on scalability. For example, if some components cannot or should not scale beyond a certain number of customers or requests, consider scaling out using stamps.
- A requirement to separate certain tenants from others. If you have customers that cannot be deployed into a multi-tenant stamp with other customers due to security concerns, they can be deployed onto their own isolated stamp.
- A need to have some tenants on different versions of your solution at the same time.
- Multi-region applications where each tenant's data and traffic should be directed to a specific region.
- A desire to achieve resiliency during outages. As stamps are independent of one another, if an outage affects a single stamp then the tenants deployed to other stamps should not be affected. This isolation helps to contain the 'blast radius' of an incident or outage.

This pattern is not suitable for:

- Simple solutions that do not need to scale to a high degree.
- Systems that can be easily scaled out or up within a single instance, such as by increasing the size of the application layer or by increasing the reserved capacity for databases and the storage tier.
- Solutions in which data should be replicated across all deployed instances. Consider the geode pattern for this scenario.
- Solutions in which only some components need to be scaled, but not others. For example, consider whether your solution could be scaled by sharding the data store rather than deploying a new copy of all of the solution components.
- Solutions comprised solely of static content, such as a front-end JavaScript application. Consider storing such content in a storage account and using Azure CDN.

Supporting technologies

- Infrastructure as code. For example, Resource Manager templates, Azure CLI, Terraform, PowerShell, Bash.
- Deployment Manager, which can orchestrate deployments of a solution across multiple stamps.
- Azure Front Door, which can route traffic to a specific stamp or to a traffic routing service.

Example

The following example deploys multiple stamps of a simple PaaS solution, with an app service and a SQL Database in each stamp. While stamps can be configured in any region that support the services deployed in the template, for illustration purposes this template deploys two stamps within the West US 2 region and a further stamp in the West Europe region. Within a stamp, the app service receives its own public DNS hostname and it can receive connections independently of all other stamps.

> **WARNING**
>
> The example below uses a SQL Server administrator account. It's generally not a good practice to use an administrative account from your application. For a real application, consider using a managed identity to connect from your application to a SQL database, or use a least-privilege account.

Click the link below to deploy the solution.

Deploy to Azure

> **NOTE**
> There are alternative approaches to deploying stamps with a Resource Manager template, including using nested templates or linked templates to decouple the definition of each stamp from the iteration required to deploy multiple copies.

Example traffic routing approach

The following example deploys an implementation of a traffic routing solution that could be used with a set of deployment stamps for a hypothetical API application. To allow for geographical distribution of incoming requests, Front Door is deployed alongside multiple instances of API Management on the consumption tier. Each API Management instance reads the tenant ID from the request URL and then looks up the relevant stamp for the request from a geo-distributed Cosmos DB data store. The request is then forwarded to the relevant back-end stamp.

Click the link below to deploy the solution.

Deploy to Azure

Related guidance

- Sharding can be used as another, simpler, approach to scale out your data tier. Stamps implicitly shard their data, but sharding does not require a Deployment Stamp. For more information, see the Sharding pattern.
- If a traffic routing service is deployed, the Gateway Routing and Gateway Offloading patterns can be used together to make the best use of this component.

Event Sourcing pattern

11/19/2021 • 14 minutes to read • Edit Online

Instead of storing just the current state of the data in a domain, use an append-only store to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects. This can simplify tasks in complex domains, by avoiding the need to synchronize the data model and the business domain, while improving performance, scalability, and responsiveness. It can also provide consistency for transactional data, and maintain full audit trails and history that can enable compensating actions.

Context and problem

Most applications work with data, and the typical approach is for the application to maintain the current state of the data by updating it as users work with it. For example, in the traditional create, read, update, and delete (CRUD) model a typical data process is to read data from the store, make some modifications to it, and update the current state of the data with the new values—often by using transactions that lock the data.

The CRUD approach has some limitations:

- CRUD systems perform update operations directly against a data store, which can slow down performance and responsiveness, and limit scalability, due to the processing overhead it requires.
- In a collaborative domain with many concurrent users, data update conflicts are more likely because the update operations take place on a single item of data.
- Unless there's an additional auditing mechanism that records the details of each operation in a separate log, history is lost.

Solution

The Event Sourcing pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. Application code sends a series of events that imperatively describe each action that has occurred on the data to the event store, where they're persisted. Each event represents a set of changes to the data (such as `AddedItemToOrder`).

The events are persisted in an event store that acts as the system of record (the authoritative data source) about the current state of the data. The event store typically publishes these events so that consumers can be notified and can handle them if needed. Consumers could, for example, initiate tasks that apply the operations in the events to other systems, or perform any other associated action that's required to complete the operation. Notice that the application code that generates the events is decoupled from the systems that subscribe to the events.

Typical uses of the events published by the event store are to maintain materialized views of entities as actions in the application change them, and for integration with external systems. For example, a system can maintain a materialized view of all customer orders that's used to populate parts of the UI. As the application adds new orders, adds or removes items on the order, and adds shipping information, the events that describe these changes can be handled and used to update the materialized view.

In addition, at any point it's possible for applications to read the history of events, and use it to materialize the current state of an entity by playing back and consuming all the events related to that entity. This can occur on demand to materialize a domain object when handling a request, or through a scheduled task so that the state of the entity can be stored as a materialized view to support the presentation layer.

The figure shows an overview of the pattern, including some of the options for using the event stream such as creating a materialized view, integrating events with external applications and systems, and replaying events to create projections of the current state of specific entities.

The Event Sourcing pattern provides the following advantages:

- Events are immutable and can be stored using an append-only operation. The user interface, workflow, or process that initiated an event can continue, and tasks that handle the events can run in the background. This, combined with the fact that there's no contention during the processing of transactions, can vastly improve performance and scalability for applications, especially for the presentation level or user interface.
- Events are simple objects that describe some action that occurred, together with any associated data required to describe the action represented by the event. Events don't directly update a data store. They're simply recorded for handling at the appropriate time. This can simplify implementation and management.
- Events typically have meaning for a domain expert, whereas object-relational impedance mismatch can make complex database tables hard to understand. Tables are artificial constructs that represent the current state of the system, not the events that occurred.
- Event sourcing can help prevent concurrent updates from causing conflicts because it avoids the requirement to directly update objects in the data store. However, the domain model must still be designed to protect itself from requests that might result in an inconsistent state.
- The append-only storage of events provides an audit trail that can be used to monitor actions taken against a data store, regenerate the current state as materialized views or projections by replaying the events at any time, and assist in testing and debugging the system. In addition, the requirement to use compensating events to cancel changes provides a history of changes that were reversed, which wouldn't be the case if the model simply stored the current state. The list of events can also be used to analyze application performance and detect user behavior trends, or to obtain other useful business information.
- The event store raises events, and tasks perform operations in response to those events. This decoupling of the tasks from the events provides flexibility and extensibility. Tasks know about the type of event and

the event data, but not about the operation that triggered the event. In addition, multiple tasks can handle each event. This enables easy integration with other services and systems that only listen for new events raised by the event store. However, the event sourcing events tend to be very low level, and it might be necessary to generate specific integration events instead.

> Event sourcing is commonly combined with the CQRS pattern by performing the data management tasks in response to the events, and by materializing views from the stored events.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

The system will only be eventually consistent when creating materialized views or generating projections of data by replaying events. There's some delay between an application adding events to the event store as the result of handling a request, the events being published, and consumers of the events handling them. During this period, new events that describe further changes to entities might have arrived at the event store.

> **NOTE**
>
> See the Data Consistency Primer for information about eventual consistency.

The event store is the permanent source of information, and so the event data should never be updated. The only way to update an entity to undo a change is to add a compensating event to the event store. If the format (rather than the data) of the persisted events needs to change, perhaps during a migration, it can be difficult to combine existing events in the store with the new version. It might be necessary to iterate through all the events making changes so they're compliant with the new format, or add new events that use the new format. Consider using a version stamp on each version of the event schema to maintain both the old and the new event formats.

Multi-threaded applications and multiple instances of applications might be storing events in the event store. The consistency of events in the event store is vital, as is the order of events that affect a specific entity (the order that changes occur to an entity affects its current state). Adding a timestamp to every event can help to avoid issues. Another common practice is to annotate each event resulting from a request with an incremental identifier. If two actions attempt to add events for the same entity at the same time, the event store can reject an event that matches an existing entity identifier and event identifier.

There's no standard approach, or existing mechanisms such as SQL queries, for reading the events to obtain information. The only data that can be extracted is a stream of events using an event identifier as the criteria. The event ID typically maps to individual entities. The current state of an entity can be determined only by replaying all of the events that relate to it against the original state of that entity.

The length of each event stream affects managing and updating the system. If the streams are large, consider creating snapshots at specific intervals such as a specified number of events. The current state of the entity can be obtained from the snapshot and by replaying any events that occurred after that point in time. For more information about creating snapshots of data, see Primary-Subordinate Snapshot Replication.

Even though event sourcing minimizes the chance of conflicting updates to the data, the application must still be able to deal with inconsistencies that result from eventual consistency and the lack of transactions. For example, an event that indicates a reduction in stock inventory might arrive in the data store while an order for that item is being placed, resulting in a requirement to reconcile the two operations either by advising the customer or creating a back order.

Event publication might be *at least once*, and so consumers of the events must be idempotent. They must not reapply the update described in an event if the event is handled more than once. For example, if multiple instances of a consumer maintain and aggregate an entity's property, such as the total number of orders placed, only one must succeed in incrementing the aggregate when an order placed event occurs. While this isn't a key

characteristic of event sourcing, it's the usual implementation decision.

When to use this pattern

Use this pattern in the following scenarios:

- When you want to capture intent, purpose, or reason in the data. For example, changes to a customer entity can be captured as a series of specific event types, such as *Moved home*, *Closed account*, or *Deceased*.
- When it's vital to minimize or completely avoid the occurrence of conflicting updates to data.
- When you want to record events that occur, and be able to replay them to restore the state of a system, roll back changes, or keep a history and audit log. For example, when a task involves multiple steps you might need to execute actions to revert updates and then replay some steps to bring the data back into a consistent state.
- When using events is a natural feature of the operation of the application, and requires little additional development or implementation effort.
- When you need to decouple the process of inputting or updating data from the tasks required to apply these actions. This might be to improve UI performance, or to distribute events to other listeners that take action when the events occur. For example, integrating a payroll system with an expense submission website so that events raised by the event store in response to data updates made in the website are consumed by both the website and the payroll system.
- When you want flexibility to be able to change the format of materialized models and entity data if requirements change, or—when used in conjunction with CQRS—you need to adapt a read model or the views that expose the data.
- When used in conjunction with CQRS, and eventual consistency is acceptable while a read model is updated, or the performance impact of rehydrating entities and data from an event stream is acceptable.

This pattern might not be useful in the following situations:

- Small or simple domains, systems that have little or no business logic, or nondomain systems that naturally work well with traditional CRUD data management mechanisms.
- Systems where consistency and real-time updates to the views of the data are required.
- Systems where audit trails, history, and capabilities to roll back and replay actions are not required.
- Systems where there's only a very low occurrence of conflicting updates to the underlying data. For example, systems that predominantly add data rather than updating it.

Example

A conference management system needs to track the number of completed bookings for a conference so that it can check whether there are seats still available when a potential attendee tries to make a booking. The system could store the total number of bookings for a conference in at least two ways:

- The system could store the information about the total number of bookings as a separate entity in a database that holds booking information. As bookings are made or canceled, the system could increment or decrement this number as appropriate. This approach is simple in theory, but can cause scalability issues if a large number of attendees are attempting to book seats during a short period of time. For example, in the last day or so prior to the booking period closing.
- The system could store information about bookings and cancellations as events held in an event store. It

could then calculate the number of seats available by replaying these events. This approach can be more scalable due to the immutability of events. The system only needs to be able to read data from the event store, or append data to the event store. Event information about bookings and cancellations is never modified.

The following diagram illustrates how the seat reservation subsystem of the conference management system might be implemented using event sourcing.

The sequence of actions for reserving two seats is as follows:

1. The user interface issues a command to reserve seats for two attendees. The command is handled by a separate command handler. A piece of logic that is decoupled from the user interface and is responsible for handling requests posted as commands.

2. An aggregate containing information about all reservations for the conference is constructed by querying the events that describe bookings and cancellations. This aggregate is called `SeatAvailability`, and is contained within a domain model that exposes methods for querying and modifying the data in the aggregate.

> Some optimizations to consider are using snapshots (so that you don't need to query and replay the full list of events to obtain the current state of the aggregate), and maintaining a cached copy of the

> aggregate in memory.

3. The command handler invokes a method exposed by the domain model to make the reservations.
4. The `SeatAvailability` aggregate records an event containing the number of seats that were reserved. The next time the aggregate applies events, all the reservations will be used to compute how many seats remain.
5. The system appends the new event to the list of events in the event store.

If a user cancels a seat, the system follows a similar process except the command handler issues a command that generates a seat cancellation event and appends it to the event store.

As well as providing more scope for scalability, using an event store also provides a complete history, or audit trail, of the bookings and cancellations for a conference. The events in the event store are the accurate record. There is no need to persist aggregates in any other way because the system can easily replay the events and restore the state to any point in time.

> You can find more information about this example in Introducing Event Sourcing.

Next steps

- Object-relational impedance mismatch
- Martin Fowler's blog:
  - Event Sourcing
  - Snapshot on Martin Fowler's Enterprise Application Architecture website

Related guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Command and Query Responsibility Segregation (CQRS) pattern. The write store that provides the permanent source of information for a CQRS implementation is often based on an implementation of the Event Sourcing pattern. Describes how to segregate the operations that read data in an application from the operations that update data by using separate interfaces.
- Materialized View pattern. The data store used in a system based on event sourcing is typically not well suited to efficient querying. Instead, a common approach is to generate prepopulated views of the data at regular intervals, or when the data changes. Shows how this can be done.
- Compensating Transaction pattern. The existing data in an event sourcing store is not updated, instead new entries are added that transition the state of entities to the new values. To reverse a change, compensating entries are used because it isn't possible to simply reverse the previous change. Describes how to undo the work that was performed by a previous operation.
- Data Consistency Primer. When using event sourcing with a separate read store or materialized views, the read data won't be immediately consistent, instead it'll be only eventually consistent. Summarizes the issues surrounding maintaining consistency over distributed data.
- Data Partitioning Guidance. Data is often partitioned when using event sourcing to improve scalability, reduce contention, and optimize performance. Describes how to divide data into discrete partitions, and the issues that can arise.

External Configuration Store pattern

11/19/2021 • 8 minutes to read • Edit Online

Move configuration information out of the application deployment package to a centralized location. This can provide opportunities for easier management and control of configuration data, and for sharing configuration data across applications and application instances.

Context and problem

The majority of application runtime environments include configuration information that's held in files deployed with the application. In some cases, it's possible to edit these files to change the application behavior after it's been deployed. However, changes to the configuration require the application be redeployed, often resulting in unacceptable downtime and other administrative overhead.

Local configuration files also limit the configuration to a single application, but sometimes it would be useful to share configuration settings across multiple applications. Examples include database connection strings, UI theme information, or the URLs of queues and storage used by a related set of applications.

It's challenging to manage changes to local configurations across multiple running instances of the application, especially in a cloud-hosted scenario. It can result in instances using different configuration settings while the update is being deployed.

In addition, updates to applications and components might require changes to configuration schemas. Many configuration systems don't support different versions of configuration information.

Solution

Store the configuration information in external storage, and provide an interface that can be used to quickly and efficiently read and update configuration settings. The type of external store depends on the hosting and runtime environment of the application. In a cloud-hosted scenario it's typically a cloud-based storage service or dedicated configuration service, but could be a hosted database or other custom system.

The backing store you choose for configuration information should have an interface that provides consistent and easy-to-use access. It should expose the information in a correctly typed and structured format. The implementation might also need to authorize users' access in order to protect configuration data, and be flexible enough to allow storage of multiple versions of the configuration (such as development, staging, or production, including multiple release versions of each one).

> Many built-in configuration systems read the data when the application starts up, and cache the data in memory to provide fast access and minimize the impact on application performance. Depending on the type of backing store used, and the latency of this store, it might be helpful to implement a caching mechanism within the external configuration store. For more information, see the Caching Guidance. The figure illustrates an overview of the External Configuration Store pattern with optional local cache.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

Choose a backing store that offers acceptable performance, high availability, robustness, and can be backed up as part of the application maintenance and administration process. In a cloud-hosted application, using a cloud storage mechanism or dedicated configuration platform service is usually a good choice to meet these requirements.

Design the schema of the backing store to allow flexibility in the types of information it can hold. Ensure that it provides for all configuration requirements such as typed data, collections of settings, multiple versions of settings, and any other features that the applications using it require. The schema should be easy to extend to support additional settings as requirements change.

Consider the physical capabilities of the backing store, how it relates to the way configuration information is stored, and the effects on performance. For example, storing an XML document containing configuration information will require either the configuration interface or the application to parse the document in order to read individual settings. It'll make updating a setting more complicated, though caching the settings can help to offset slower read performance.

Consider how the configuration interface will permit control of the scope and inheritance of configuration settings. For example, it might be a requirement to scope configuration settings at the organization, application, and the machine level. It might need to support delegation of control over access to different scopes, and to prevent or allow individual applications to override settings.

Ensure that the configuration interface can expose the configuration data in the required formats such as typed values, collections, key/value pairs, or property bags.

Consider how the configuration store interface will behave when settings contain errors, or don't exist in the backing store. It might be appropriate to return default settings and log errors. Also consider aspects such as the case sensitivity of configuration setting keys or names, the storage and handling of binary data, and the ways that null or empty values are handled.

Consider how to protect the configuration data to allow access to only the appropriate users and applications. This is likely a feature of the configuration store interface, but it's also necessary to ensure that the data in the backing store can't be accessed directly without the appropriate permission. Ensure strict separation between the permissions required to read and to write configuration data. Also consider whether you need to encrypt some or all of the configuration settings, and how this'll be implemented in the configuration store interface.

Centrally stored configurations, which change application behavior during runtime, are critically important and should be deployed, updated, and managed using the same mechanisms as deploying application code. For example, changes that can affect more than one application must be carried out using a full test and staged deployment approach to ensure that the change is appropriate for all applications that use this configuration. If

an administrator edits a setting to update one application, it could adversely impact other applications that use the same setting.

If an application caches configuration information, the application needs to be alerted if the configuration changes. It might be possible to implement an expiration policy over cached configuration data so that this information is automatically refreshed periodically and any changes picked up (and acted on).

While caching configuration data can help address transient connectivity issues with the external configuration store at application runtime, this typically doesn't solve the problem if the external store is down when the application is first starting. Ensure your application deployment pipeline can provide the last known set of configuration values in a configuration file as a fallback if your application cannot retrieve live values when it starts.

When to use this pattern

This pattern is useful for:

- Configuration settings that are shared between multiple applications and application instances, or where a standard configuration must be enforced across multiple applications and application instances.
- A standard configuration system that doesn't support all of the required configuration settings, such as storing images or complex data types.
- As a complementary store for some of the settings for applications, perhaps allowing applications to override some or all of the centrally-stored settings.
- As a way to simplify administration of multiple applications, and optionally for monitoring use of configuration settings by logging some or all types of access to the configuration store.

Custom backing store example

In a Microsoft Azure hosted application, a possible choice for storing configuration information externally is to use Azure Storage. This is resilient, offers high performance, and is replicated three times with automatic failover to offer high availability. Azure Table storage provides a key/value store with the ability to use a flexible schema for the values. Azure Blob storage provides a hierarchical, container-based store that can hold any type of data in individually named blobs.

When implementing this pattern you'd be responsible for abstracting away Azure Blob storage and exposing your settings within your applications, including checking for updates at runtime and addressing how to respond to those.

The following example shows how a simplistic configuration store could be envisioned over Blob storage to store and expose configuration information. A `BlobSettingsStore` class could abstract Blob storage for holding configuration information, and implements a simple `ISettingsStore` interface.

```
public interface ISettingsStore
{
    Task<ETag> GetVersionAsync();
    Task<Dictionary<string, string>> FindAllAsync();
}
```

This interface defines methods for retrieving configuration settings held in the configuration store and includes a version number that can be used to detect whether any configuration settings have been modified recently. A `BlobSettingsStore` class could use the `ETag` property of the blob to implement versioning. The `ETag` property is updated automatically each time a blob is written.

> By design, this simple illustration exposes all configuration settings as string values rather than typed values.

An `ExternalConfigurationManager` class could then provide a wrapper around a `BlobSettingsStore` instance. An application can use this class to retrieve configuration information. This class might use something like Microsoft Reactive Extensions to publish any changes made to the configuration while the system is running. It would also be responsible for implementing the Cache-Aside pattern for settings to provide added resiliency and performance.

Usage might look something like the following.

```
static void Main(string[] args)
{
    // Start monitoring configuration changes.
    ExternalConfiguration.Instance.StartMonitor();

    // Get a setting.
    var setting = ExternalConfiguration.Instance.GetAppSetting("someSettingKey");
    …
}
```

Using Azure App Configuration

While building a custom configuration store might be necessary in some situations, many applications can instead use Azure App Configuration. Azure App Configuration supports key-value pairs that can be namespaced. The keys are typed and are individually versioned. Azure App Configuration also supports point-in-time snapshots of configuration so that you can easily inspect or even roll back to prior configuration values. Configuration values can be exported such that a copy of the configuration can ship with your application in case the service is unreachable when the application is starting.

Client libraries

Many of these features are exposed through client libraries which integrate with the application runtime to facilitate fetching and caching values, refreshing values on change, and even handling transient outages of App Configuration Service.

RUNTIME	CLIENT LIBRARY	NOTES	QUICKSTART
.NET	Microsoft.Extensions.Configuration.AzureAppConfiguration	Provider for `Microsoft.Extensions.Configuration`	Quickstart
ASP.NET	Microsoft.Azure.AppConfiguration.AspNetCore	Provider for `Microsoft.Extensions.Configuration`	Quickstart
Azure Functions in .NET	Microsoft.Extensions.Configuration.AzureAppConfiguration	Used with Azure Function extensions to support configuration in *Startup.cs*	Quickstart
.NET Framework	Microsoft.Configuration.ConfigurationBuilders.AzureAppConfiguration	Configuration builder for `System.Configuration`	Quickstart
Java Spring	com.azure.spring > azure-spring-cloud-appconfiguration-config	Supports Spring Framework access via `ConfigurationProperties`	Quickstart

RUNTIME	CLIENT LIBRARY	NOTES	QUICKSTART
Python	azure.appconfiguration	Provides an `AzureAppConfigurationClient`	Quickstart
JavaScript/Node.js	@azure/app-configuration	Provides an `AppConfigurationClient`	Quickstart

In addition to client libraries, there are also an Azure App Configuration Sync GitHub Action and Azure App Configuration Pull & Azure App Configuration Push Azure DevOps tasks to integrate configuration steps into your build process.

Next steps

- See additional App Configuration Samples
- Learn how to integrate Azure App Configuration with Kubernetes deployments
- Learn how Azure App Configuration also can help manage feature flags

Federated Identity pattern

11/19/2021 • 7 minutes to read • Edit Online

Delegate authentication to an external identity provider. This can simplify development, minimize the requirement for user administration, and improve the user experience of the application.

Context and problem

Users typically need to work with multiple applications provided and hosted by different organizations they have a business relationship with. These users might be required to use specific (and different) credentials for each one. This can:

- **Cause a disjointed user experience**. Users often forget sign-in credentials when they have many different ones.
- **Expose security vulnerabilities**. When a user leaves the company the account must immediately be deprovisioned. It's easy to overlook this in large organizations.
- **Complicate user management**. Administrators must manage credentials for all of the users, and perform additional tasks such as providing password reminders.

Users typically prefer to use the same credentials for all these applications.

Solution

Implement an authentication mechanism that can use federated identity. Separate user authentication from the application code, and delegate authentication to a trusted identity provider. This can simplify development and allow users to authenticate using a wider range of identity providers (IdP) while minimizing the administrative overhead. It also allows you to clearly decouple authentication from authorization.

The trusted identity providers include corporate directories, on-premises federation services, other security token services (STS) provided by business partners, or social identity providers that can authenticate users who have, for example, a Microsoft, Google, Yahoo!, or Facebook account.

The figure illustrates the Federated Identity pattern when a client application needs to access a service that requires authentication. The authentication is performed by an IdP that works in concert with an STS. The IdP issues security tokens that provide information about the authenticated user. This information, referred to as claims, includes the user's identity, and might also include other information such as role membership and more granular access rights.

This model is often called claims-based access control. Applications and services authorize access to features and functionality based on the claims contained in the token. The service that requires authentication must trust the IdP. The client application contacts the IdP that performs the authentication. If the authentication is successful, the IdP returns a token containing the claims that identify the user to the STS (note that the IdP and STS can be the same service). The STS can transform and augment the claims in the token based on predefined rules, before returning it to the client. The client application can then pass this token to the service as proof of its identity.

> There might be additional security token services in the chain of trust. For example, in the scenario described later, an on-premises STS trusts another STS that is responsible for accessing an identity provider to authenticate the user. This approach is common in enterprise scenarios where there's an on-premises STS and directory.

Federated authentication provides a standards-based solution to the issue of trusting identities across diverse domains, and can support single sign-on. It's becoming more common across all types of applications, especially cloud-hosted applications, because it supports single sign-on without requiring a direct network connection to identity providers. The user doesn't have to enter credentials for every application. This increases security because it prevents the creation of credentials required to access many different applications, and it also hides the user's credentials from all but the original identity provider. Applications see just the authenticated identity information contained within the token.

Federated identity also has the major advantage that management of the identity and credentials is the responsibility of the identity provider. The application or service doesn't need to provide identity management features. In addition, in corporate scenarios, the corporate directory doesn't need to know about the user if it trusts the identity provider. This removes all the administrative overhead of managing the user identity within the directory.

Issues and considerations

Consider the following when designing applications that implement federated authentication:

- Authentication can be a single point of failure. If you deploy your application to multiple datacenters, consider deploying your identity management mechanism to the same datacenters to maintain application reliability and availability.
- Authentication tools make it possible to configure access control based on role claims contained in the authentication token. This is often referred to as role-based access control (RBAC), and it can allow a more granular level of control over access to features and resources.

- Unlike a corporate directory, claims-based authentication using social identity providers doesn't usually provide information about the authenticated user other than an email address, and perhaps a name. Some social identity providers, such as a Microsoft account, provide only a unique identifier. The application usually needs to maintain some information on registered users, and be able to match this information to the identifier contained in the claims in the token. Typically this is done through registration when the user first accesses the application, and information is then injected into the token as additional claims after each authentication.
- If there's more than one identity provider configured for the STS, it must detect which identity provider the user should be redirected to for authentication. This process is called home realm discovery. The STS might be able to do this automatically based on an email address or user name that the user provides, a subdomain of the application that the user is accessing, the user's IP address scope, or on the contents of a cookie stored in the user's browser. For example, if the user entered an email address in the Microsoft domain, such as user@live.com, the STS will redirect the user to the Microsoft account sign-in page. On later visits, the STS could use a cookie to indicate that the last sign in was with a Microsoft account. If automatic discovery can't determine the home realm, the STS will display a home realm discovery page that lists the trusted identity providers, and the user must select the one they want to use.

When to use this pattern

This pattern is useful for scenarios such as:

- **Single sign-on in the enterprise**. In this scenario you need to authenticate employees for corporate applications that are hosted in the cloud outside the corporate security boundary, without requiring them to sign in every time they visit an application. The user experience is the same as when using on-premises applications where they're authenticated when signing in to a corporate network, and from then on have access to all relevant applications without needing to sign in again.
- **Federated identity with multiple partners**. In this scenario you need to authenticate both corporate employees and business partners who don't have accounts in the corporate directory. This is common in business-to-business applications, applications that integrate with third-party services, and where companies with different IT systems have merged or shared resources.
- **Federated identity in SaaS applications**. In this scenario independent software vendors provide a ready-to-use service for multiple clients or tenants. Each tenant authenticates using a suitable identity provider. For example, business users will use their corporate credentials, while consumers and clients of the tenant will use their social identity credentials.

This pattern might not be useful in the following situations:

- All users of the application can be authenticated by one identity provider, and there's no requirement to authenticate using any other identity provider. This is typical in business applications that use a corporate directory (accessible within the application) for authentication, by using a VPN, or (in a cloud-hosted scenario) through a virtual network connection between the on-premises directory and the application.
- The application was originally built using a different authentication mechanism, perhaps with custom user stores, or doesn't have the capability to handle the negotiation standards used by claims-based technologies. Retrofitting claims-based authentication and access control into existing applications can be complex, and probably not cost effective.

Example

An organization hosts a multi-tenant software as a service (SaaS) application in Microsoft Azure. The application includes a website that tenants can use to manage the application for their own users. The application allows tenants to access the website by using a federated identity that is generated by Active Directory Federation

Services (AD FS) when a user is authenticated by that organization's own Active Directory.

The figure shows how tenants authenticate with their own identity provider (step 1), in this case AD FS. After successfully authenticating a tenant, AD FS issues a token. The client browser forwards this token to the SaaS application's federation provider, which trusts tokens issued by the tenant's AD FS, in order to get back a token that is valid for the SaaS federation provider (step 2). If necessary, the SaaS federation provider performs a transformation on the claims in the token into claims that the application recognizes (step 3) before returning the new token to the client browser. The application trusts tokens issued by the SaaS federation provider and uses the claims in the token to apply authorization rules (step 4).

Tenants won't need to remember separate credentials to access the application, and an administrator at the tenant's company can configure in its own AD FS the list of users that can access the application.

Next steps

- Microsoft Azure Active Directory
- Active Directory Domain Services
- Active Directory Federation Services
- Identity management for multitenant applications in Microsoft Azure
- Multitenant Applications in Azure

Gatekeeper pattern

11/19/2021 • 4 minutes to read • Edit Online

Protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service, validates and sanitizes requests, and passes requests and data between them. This can provide an additional layer of security, and limit the attack surface of the system.

Context and problem

Applications expose their functionality to clients by accepting and processing requests. In cloud-hosted scenarios, applications expose endpoints clients connect to, and typically include the code to handle the requests from clients. This code performs authentication and validation, some or all request processing, and is likely to accesses storage and other services on behalf of the client.

If a malicious user is able to compromise the system and gain access to the application's hosting environment, the security mechanisms it uses such as credentials and storage keys, and the services and data it accesses, are exposed. As a result, the malicious user can gain unrestrained access to sensitive information and other services.

Solution

To minimize the risk of clients gaining access to sensitive information and services, decouple hosts or tasks that expose public endpoints from the code that processes requests and accesses storage. You can achieve this by using a façade or a dedicated task that interacts with clients and then hands off the request—perhaps through a decoupled interface—to the hosts or tasks that'll handle the request. The figure provides a high-level overview of this pattern.

The gatekeeper pattern can be used to simply protect storage, or it can be used as a more comprehensive façade to protect all of the functions of the application. The important factors are:

- **Controlled validation**. The gatekeeper validates all requests, and rejects those that don't meet validation requirements.
- **Limited risk and exposure**. The gatekeeper doesn't have access to the credentials or keys used by the trusted host to access storage and services. If the gatekeeper is compromised, the attacker doesn't get access to these credentials or keys.
- **Appropriate security**. The gatekeeper runs in a limited privilege mode, while the rest of the application runs in the full trust mode required to access storage and services. If the gatekeeper is compromised, it can't directly access the application services or data.

This pattern acts like a firewall in a typical network topography. It allows the gatekeeper to examine requests and make a decision about whether to pass the request on to the trusted host that performs the required tasks. This

decision typically requires the gatekeeper to validate and sanitize the request content before passing it on to the trusted host.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Ensure that the trusted hosts the gatekeeper passes requests to expose only internal or protected endpoints, and connect only to the gatekeeper. The trusted hosts shouldn't expose any external endpoints or interfaces.
- The gatekeeper must run in a limited privilege mode. Typically this means running the gatekeeper and the trusted host in separate hosted services or virtual machines.
- The gatekeeper shouldn't perform any processing related to the application or services, or access any data. Its function is purely to validate and sanitize requests. The trusted hosts might need to perform additional validation of requests, but the core validation should be performed by the gatekeeper.
- Use a secure communication channel (HTTPS, SSL, or TLS) between the gatekeeper and the trusted hosts or tasks where this is possible. However, some hosting environments don't support HTTPS on internal endpoints.
- Adding the extra layer to the application to implement the gatekeeper pattern is likely to have some impact on performance due to the additional processing and network communication it requires.
- The gatekeeper instance could be a single point of failure. To minimize the impact of a failure, consider deploying additional instances and using an autoscaling mechanism to ensure capacity to maintain availability.

When to use this pattern

This pattern is useful for:

- Applications that handle sensitive information, expose services that must have a high degree of protection from malicious attacks, or perform mission-critical operations that shouldn't be disrupted.
- Distributed applications where it's necessary to perform request validation separately from the main tasks, or to centralize this validation to simplify maintenance and administration.

Example

In a cloud-hosted scenario, this pattern can be implemented by decoupling the gatekeeper role or virtual machine from the trusted roles and services in an application. Do this by using an internal endpoint, a queue, or storage as an intermediate communication mechanism. The figure illustrates using an internal endpoint.

Related guidance

The Valet Key pattern might also be relevant when implementing the Gatekeeper pattern. When communicating

between the Gatekeeper and trusted roles, it's a good practice to enhance security by using keys or tokens that limit permissions for accessing resources. The pattern describes how to use a token or key that provides clients with restricted, direct access to a specific resource or service.

Gateway Aggregation pattern

11/19/2021 • 3 minutes to read • Edit Online

Use a gateway to aggregate multiple individual requests into a single request. This pattern is useful when a client must make multiple calls to different backend systems to perform an operation.

Context and problem

To perform a single task, a client may have to make multiple calls to various backend services. An application that relies on many services to perform a task must expend resources on each request. When any new feature or service is added to the application, additional requests are needed, further increasing resource requirements and network calls. This chattiness between a client and a backend can adversely impact the performance and scale of the application. Microservice architectures have made this problem more common, as applications built around many smaller services naturally have a higher amount of cross-service calls.

In the following diagram, the client sends requests to each service (1,2,3). Each service processes the request and sends the response back to the application (4,5,6). Over a cellular network with typically high latency, using individual requests in this manner is inefficient and could result in broken connectivity or incomplete requests. While each request may be done in parallel, the application must send, wait, and process data for each request, all on separate connections, increasing the chance of failure.

Solution

Use a gateway to reduce chattiness between the client and the services. The gateway receives client requests, dispatches requests to the various backend systems, and then aggregates the results and sends them back to the requesting client.

This pattern can reduce the number of requests that the application makes to backend services, and improve application performance over high-latency networks.

In the following diagram, the application sends a request to the gateway (1). The request contains a package of additional requests. The gateway decomposes these and processes each request by sending it to the relevant service (2). Each service returns a response to the gateway (3). The gateway combines the responses from each service and sends the response to the application (4). The application makes a single request and receives only a single response from the gateway.

Issues and considerations

- The gateway should not introduce service coupling across the backend services.
- The gateway should be located near the backend services to reduce latency as much as possible.
- The gateway service may introduce a single point of failure. Ensure the gateway is properly designed to meet your application's availability requirements.
- The gateway may introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can be scaled to meet your anticipated growth.
- Perform load testing against the gateway to ensure you don't introduce cascading failures for services.
- Implement a resilient design, using techniques such as bulkheads, circuit breaking, retry, and timeouts.
- If one or more service calls takes too long, it may be acceptable to timeout and return a partial set of data. Consider how your application will handle this scenario.
- Use asynchronous I/O to ensure that a delay at the backend doesn't cause performance issues in the application.
- Implement distributed tracing using correlation IDs to track each individual call.
- Monitor request metrics and response sizes.
- Consider returning cached data as a failover strategy to handle failures.
- Instead of building aggregation into the gateway, consider placing an aggregation service behind the gateway. Request aggregation will likely have different resource requirements than other services in the gateway and may impact the gateway's routing and offloading functionality.

When to use this pattern

Use this pattern when:

- A client needs to communicate with multiple backend services to perform an operation.
- The client may use networks with significant latency, such as cellular networks.

This pattern may not be suitable when:

- You want to reduce the number of calls between a client and a single service across multiple operations. In that scenario, it may be better to add a batch operation to the service.
- The client or application is located near the backend services and latency is not a significant factor.

Example

The following example illustrates how to create a simple a gateway aggregation NGINX service using Lua.

```
worker_processes  4;

events {
  worker_connections 1024;
}

http {
  server {
    listen 80;

    location = /batch {
      content_by_lua '
        ngx.req.read_body()

        -- read json body content
        local cjson = require "cjson"
        local batch = cjson.decode(ngx.req.get_body_data())["batch"]

        -- create capture_multi table
        local requests = {}
        for i, item in ipairs(batch) do
          table.insert(requests, {item.relative_url, { method = ngx.HTTP_GET}})
        end

        -- execute batch requests in parallel
        local results = {}
        local resps = { ngx.location.capture_multi(requests) }
        for i, res in ipairs(resps) do
          table.insert(results, {status = res.status, body = cjson.decode(res.body), header = res.header})
        end

        ngx.say(cjson.encode({results = results}))
      ';
    }

    location = /service1 {
      default_type application/json;
      echo '{"attr1":"val1"}';
    }

    location = /service2 {
      default_type application/json;
      echo '{"attr2":"val2"}';
    }
  }
}
```

Related guidance

- Backends for Frontends pattern
- Gateway Offloading pattern
- Gateway Routing pattern

Gateway Offloading pattern

11/19/2021 • 3 minutes to read • Edit Online

Offload shared or specialized service functionality to a gateway proxy. This pattern can simplify application development by moving shared service functionality, such as the use of SSL certificates, from other parts of the application into the gateway.

Context and problem

Some features are commonly used across multiple services, and these features require configuration, management, and maintenance. A shared or specialized service that is distributed with every application deployment increases the administrative overhead and increases the likelihood of deployment error. Any updates to a shared feature must be deployed across all services that share that feature.

Properly handling security issues (token validation, encryption, SSL certificate management) and other complex tasks can require team members to have highly specialized skills. For example, a certificate needed by an application must be configured and deployed on all application instances. With each new deployment, the certificate must be managed to ensure that it does not expire. Any common certificate that is due to expire must be updated, tested, and verified on every application deployment.

Other common services such as authentication, authorization, logging, monitoring, or throttling can be difficult to implement and manage across a large number of deployments. It may be better to consolidate this type of functionality, in order to reduce overhead and the chance of errors.

Solution

Offload some features into a gateway, particularly cross-cutting concerns such as certificate management, authentication, SSL termination, monitoring, protocol translation, or throttling.

The following diagram shows a gateway that terminates inbound SSL connections. It requests data on behalf of the original requestor from any HTTP server upstream of the gateway.

Benefits of this pattern include:

- Simplify the development of services by removing the need to distribute and maintain supporting resources, such as web server certificates and configuration for secure websites. Simpler configuration results in easier management and scalability and makes service upgrades simpler.
- Allow dedicated teams to implement features that require specialized expertise, such as security. This allows your core team to focus on the application functionality, leaving these specialized but cross-cutting concerns to the relevant experts.
- Provide some consistency for request and response logging and monitoring. Even if a service is not

correctly instrumented, the gateway can be configured to ensure a minimum level of monitoring and logging.

Issues and considerations

- Ensure the gateway is highly available and resilient to failure. Avoid single points of failure by running multiple instances of your gateway.
- Ensure the gateway is designed for the capacity and scaling requirements of your application and endpoints. Make sure the gateway does not become a bottleneck for the application and is sufficiently scalable.
- Only offload features that are used by the entire application, such as security or data transfer.
- Business logic should never be offloaded to the gateway.
- If you need to track transactions, consider generating correlation IDs for logging purposes.

When to use this pattern

Use this pattern when:

- An application deployment has a shared concern such as SSL certificates or encryption.
- A feature that is common across application deployments that may have different resource requirements, such as memory resources, storage capacity or network connections.
- You wish to move the responsibility for issues such as network security, throttling, or other network boundary concerns to a more specialized team.

This pattern may not be suitable if it introduces coupling across services.

Example

Using Nginx as the SSL offload appliance, the following configuration terminates an inbound SSL connection and distributes the connection to one of three upstream HTTP servers.

```
upstream iis {
        server  10.3.0.10    max_fails=3    fail_timeout=15s;
        server  10.3.0.20    max_fails=3    fail_timeout=15s;
        server  10.3.0.30    max_fails=3    fail_timeout=15s;
}

server {
        listen 443;
        ssl on;
        ssl_certificate /etc/nginx/ssl/domain.cer;
        ssl_certificate_key /etc/nginx/ssl/domain.key;

        location / {
                set $targ iis;
                proxy_pass http://$targ;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header X-Forwarded-Proto https;
proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header Host $host;
        }
}
```

On Azure, this can be achieved by setting up SSL termination on Application Gateway.

Related guidance

- Backends for Frontends pattern

- Gateway Aggregation pattern
- Gateway Routing pattern

Gateway Routing pattern

11/19/2021 • 3 minutes to read • Edit Online

Route requests to multiple services using a single endpoint. This pattern is useful when you wish to expose multiple services on a single endpoint and route to the appropriate service based on the request.

Context and problem

When a client needs to consume multiple services, setting up a separate endpoint for each service and having the client manage each endpoint can be challenging. For example, an e-commerce application might provide services such as search, reviews, cart, checkout, and order history. Each service has a different API that the client must interact with, and the client must know about each endpoint in order to connect to the services. If an API changes, the client must be updated as well. If you refactor a service into two or more separate services, the code must change in both the service and the client.

Solution

Place a gateway in front of a set of applications, services, or deployments. Use application Layer 7 routing to route the request to the appropriate instances.

With this pattern, the client application only needs to know about and communicate with a single endpoint. If a service is consolidated or decomposed, the client does not necessarily require updating. It can continue making requests to the gateway, and only the routing changes.

A gateway also lets you abstract backend services from the clients, allowing you to keep client calls simple while enabling changes in the backend services behind the gateway. Client calls can be routed to whatever service or services need to handle the expected client behavior, allowing you to add, split, and reorganize services behind the gateway without changing the client.

This pattern can also help with deployment, by allowing you to manage how updates are rolled out to users. When a new version of your service is deployed, it can be deployed in parallel with the existing version. Routing lets you control what version of the service is presented to the clients, giving you the flexibility to use various release strategies, whether incremental, parallel, or complete rollouts of updates. Any issues discovered after the new service is deployed can be quickly reverted by making a configuration change at the gateway, without affecting clients.

Issues and considerations

- The gateway service may introduce a single point of failure. Ensure it is properly designed to meet your availability requirements. Consider resiliency and fault tolerance capabilities when implementing.
- The gateway service may introduce a bottleneck. Ensure the gateway has adequate performance to handle load and can easily scale in line with your growth expectations.
- Perform load testing against the gateway to ensure you don't introduce cascading failures for services.
- Gateway routing is level 7. It can be based on IP, port, header, or URL.

When to use this pattern

Use this pattern when:

- A client needs to consume multiple services that can be accessed behind a gateway.
- You wish to simplify client applications by using a single endpoint.
- You need to route requests from externally addressable endpoints to internal virtual endpoints, such as exposing ports on a VM to cluster virtual IP addresses.

This pattern may not be suitable when you have a simple application that uses only one or two services.

Example

Using Nginx as the router, the following is a simple example configuration file for a server that routes requests

for applications residing on different virtual directories to different machines at the back end.

```
server {
    listen 80;
    server_name domain.com;

    location /app1 {
        proxy_pass http://10.0.3.10:80;
    }

    location /app2 {
        proxy_pass http://10.0.3.20:80;
    }

    location /app3 {
        proxy_pass http://10.0.3.30:80;
    }
}
```

On Azure, multiple services can be set up behind an Application Gateway instance, which provides layer-7 routing.

Related guidance

- Backends for Frontends pattern
- Gateway Aggregation pattern
- Gateway Offloading pattern

Geode pattern

11/19/2021 • 7 minutes to read • Edit Online

The Geode pattern involves deploying a collection of backend services into a set of **ge**ographical n**ode**s, each of which can service any request for any client in any region. This pattern allows serving requests in an *active-active* style, improving latency and increasing availability by distributing request processing around the globe.

Context and problem

Many large-scale services have specific challenges around geo-availability and scale. Classic designs often *bring the data to the compute* by storing data in a remote SQL server that serves as the compute tier for that data, relying on scale-up for growth.

The classic approach may present a number of challenges:

- Network latency issues for users coming from the other side of the globe to connect to the hosting endpoint
- Traffic management for demand bursts that can overwhelm the services in a single region
- Cost-prohibitive complexity of deploying copies of app infrastructure into multiple regions for a 24x7 service

Modern cloud infrastructure has evolved to enable geographic load balancing of front-end services, while allowing for geographic replication of backend services. For availability and performance, getting data closer to the user is good. When data is geo-distributed across a far-flung user base, the geo-distributed datastores should also be colocated with the compute resources that process the data. The geode pattern *brings the compute to the data*.

Solution

Deploy the service into a number of satellite deployments spread around the globe, each of which is called a *geode*. The geode pattern harnesses key features of Azure to route traffic via the shortest path to a nearby geode, which improves latency and performance. Each geode is behind a global load balancer, and uses a geo-replicated read-write service like Azure Cosmos DB to host the data plane, ensuring cross-geode data consistency. Data replication services ensure that data stores are identical across geodes, so *all* requests can be served from *all* geodes.

The key difference between a deployment stamp and a geode is that geodes never exist in isolation. There should always be more than one geode in a production platform.

Geodes have the following characteristics:

- Consist of a collection of disparate types of resources, often defined in a template.
- Have no dependencies outside of the geode footprint and are self-contained. No geode is dependent on another to operate, and if one dies, the others continue to operate.
- Are loosely coupled via an edge network and replication backplane. For example, you can use Azure Traffic Manager or Azure Front Door for fronting the geodes, while Azure Cosmos DB can act as the replication backplane. Geodes are not the same as clusters because they share a replication backplane, so the platform takes care of quorum issues.

The geode pattern occurs in big data architectures that use commodity hardware to process data colocated on the same machine, and MapReduce to consolidate results across machines. Another usage is near-edge compute, which brings compute closer to the intelligent edge of the network to reduce response time.

Services can use this pattern over dozens or hundreds of geodes. Furthermore, the resiliency of the whole solution increases with each added geode, since any geodes can take over if a regional outage takes one or more geodes offline.

It's also possible to augment local availability techniques, such as availability zones or paired regions, with the geode pattern for global availability. This increases complexity, but is useful if your architecture is underpinned by a storage engine such as blob storage that can only replicate to a paired region. You can deploy geodes into an intra-zone, zonal, or regional footprint, with a mind to regulatory or latency constraints on location.

Issues and considerations

Use the following techniques and technologies to implement this pattern:

- Modern DevOps practices and tools to produce and rapidly deploy identical geodes across a large number of regions or instances.
- Autoscaling to scale out compute and database throughput instances within a geode. Each geode individually scales out, within the common backplane constraints.
- A front-end service like Azure Front Door that does dynamic content acceleration, split TCP, and Anycast routing.
- A replicating data store like Azure Cosmos DB to control data consistency.
- Serverless technologies where possible, to reduce always-on deployment cost, especially when load is frequently rebalanced around the globe. This strategy allows for many geodes to be deployed with minimal additional investment. Serverless and consumption-based billing technologies reduce waste and cost from

duplicate geo-distributed deployments.

- API Management is not required to implement the design pattern, but can be added to each geode that fronts the region's Azure Function App to provide a more robust API layer, enabling the implementation of additional functionality like rate limiting, for instance.

Consider the following points when deciding how to implement this pattern:

- Choose whether to process data locally in each region, or to distribute aggregations in a single geode and replicate the result across the globe. The Azure Cosmos DB change feed processor offers this granular control using its *lease container* concept, and the *leasecollectionprefix* in the corresponding Azure Functions binding. Each approach has distinct advantages and drawbacks.
- Geodes can work in tandem, using the Azure Cosmos DB change feed and a real-time communication platform like SignalR. Geodes can communicate with remote users via other geodes in a mesh pattern, without knowing or caring where the remote user is located.
- This design pattern implicitly decouples everything, resulting in an ultra-highly distributed and decoupled architecture. Consider how to track different components of the same request as they might execute asynchronously on different instances. A proper monitoring strategy is crucial. Both Azure Front Door and Cosmos DB can be easily integrated with Log Analytics and Azure Functions should be deployed alongside Application Insights to provide a robust monitoring system at each component in the architecture.
- Distributed deployments have a greater number of secrets and ingress points that require property security measures. Key Vault provides a secure layer for secret management and each layer within the API architecture should be properly secured so that the only ingress point for the API is the front-end service like Azure Front Door. The Cosmos DB should restrict traffic to the Azure Function Apps, and the Function apps to Azure Front Door using Azure Active Directory or practices like IP restriction.
- Performance is drastically affected by the number of geodes that are deployed and the specific App Service Plans applied to the API technology in each geode. Deployment of additional geodes or movement towards premium tiers come with increased costs for the additional memory and compute, but do not do so on a per transaction basis. Consider load testing the API architecture once deployed and contrast increasing the numbers of geodes with increasing the pricing tier so that the most cost-efficient model is used for your needs.
- Determine the availability requirements for your data. Cosmos DB has optional flags for enabling multi-region write, availability zones, and more. These increase the availability for the Cosmos DB instance and creates a more resilient data layer, but come with additional costs.
- Azure offers a variety of load balancers that provide different functionalities for distribution of traffic. Use the decision tree to help select the right option for your API's front end.

When to use this pattern

Use this pattern:

- To implement a high-scale platform that has users distributed over a wide area.
- For any service that requires extreme availability and resilience characteristics, because services based on the geode pattern can survive the loss of multiple service regions at the same time.

This pattern might not be suitable for

- Architectures that have constraints so that all geodes can't be equal for data storage. For example, there may be data residency requirements, an application that needs to maintain temporary state for a particular session, or a heavy weighting of requests towards a single region. In this case, consider using deployment stamps in combination with a global routing plane that is aware of where a user's data sits, such as the traffic routing component described within the deployment stamps pattern.
- Situations where there's no geographical distribution required. Instead, consider availability zones and paired regions for clustering.

- Situations where a legacy platform needs to be retrofitted. This pattern works for cloud-native development only, and can be difficult to retrofit.
- Simple architectures and requirements, where geo-redundancy and geo-distribution aren't required or advantageous.

Examples

- Windows Active Directory implements an early variant of this pattern. Multi-primary replication means all updates and requests can in theory be served from all serviceable nodes, but Flexible Single Master Operation (FSMO) roles mean that all geodes aren't equal.
- The geode pattern accelerator on GitHub showcases this design pattern in practice and is designed to help developers implement it with real-world APIs.
- The globally distributed applications using Cosmos DB article examines a geographical based deployment that utilizes Traffic Manager for load balancing and Azure App Service to host the API code.
- A QnA sample application on GitHub showcases this design pattern in practice.
- Geode Cache over SAP OData APIs: A sample OData API Geode set backed by Cosmos as a globally accelerated data cache for SAP Retail applications.

Health Endpoint Monitoring pattern

11/19/2021 • 11 minutes to read • Edit Online

Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals. This can help to verify that applications and services are performing correctly.

Context and problem

It's a good practice, and often a business requirement, to monitor web applications and back-end services, to ensure they're available and performing correctly. However, it's sometimes more difficult to monitor services running in the cloud than it is to monitor on-premises services. For example, you don't have full control of the hosting environment, and the services typically depend on other services provided by platform vendors and others.

There are many factors that affect cloud-hosted applications such as network latency, the performance and availability of the underlying compute and storage systems, and the network bandwidth between them. The service can fail entirely or partially due to any of these factors. Therefore, you must verify at regular intervals that the service is performing correctly to ensure the required level of availability, which might be part of your service level agreement (SLA).

Solution

Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks, and return an indication of its status.

A health monitoring check typically combines two factors:

- The checks (if any) performed by the application or service in response to the request to the health verification endpoint.
- Analysis of the results by the tool or framework that performs the health verification check.

The response code indicates the status of the application and, optionally, any components or services it uses. The latency or response time check is performed by the monitoring tool or framework. The figure provides an overview of the pattern.

Other checks that might be carried out by the health monitoring code in the application include:

- Checking cloud storage or a database for availability and response time.
- Checking other resources or services located in the application, or located elsewhere but used by the application.

Services and tools are available that monitor web applications by submitting a request to a configurable set of endpoints, and evaluating the results against a set of configurable rules. It's relatively easy to create a service endpoint whose sole purpose is to perform some functional tests on the system.

Typical checks that can be performed by the monitoring tools include:

- Validating the response code. For example, an HTTP response of 200 (OK) indicates that the application responded without error. The monitoring system might also check for other response codes to give more comprehensive results.
- Checking the content of the response to detect errors, even when a 200 (OK) status code is returned. This can detect errors that affect only a section of the returned web page or service response. For example, checking the title of a page or looking for a specific phrase that indicates the correct page was returned.
- Measuring the response time, which indicates a combination of the network latency and the time that the application took to execute the request. An increasing value can indicate an emerging problem with the application or network.
- Checking resources or services located outside the application, such as a content delivery network used by the application to deliver content from global caches.
- Checking for expiration of SSL certificates.
- Measuring the response time of a DNS lookup for the URL of the application to measure DNS latency and DNS failures.
- Validating the URL returned by the DNS lookup to ensure correct entries. This can help to avoid malicious request redirection through a successful attack on the DNS server.

It's also useful, where possible, to run these checks from different on-premises or hosted locations to measure and compare response times. Ideally you should monitor applications from locations that are close to customers to get an accurate view of the performance from each location. In addition to providing a more robust checking mechanism, the results can help you decide on the deployment location for the application—and whether to deploy it in more than one datacenter.

Tests should also be run against all the service instances that customers use to ensure the application is working

correctly for all customers. For example, if customer storage is spread across more than one storage account, the monitoring process should check all of these.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

How to validate the response. For example, is just a single 200 (OK) status code sufficient to verify the application is working correctly? While this provides the most basic measure of application availability, and is the minimum implementation of this pattern, it provides little information about the operations, trends, and possible upcoming issues in the application.

The number of endpoints to expose for an application. One approach is to expose at least one endpoint for the core services that the application uses and another for lower priority services, allowing different levels of importance to be assigned to each monitoring result. Also consider exposing more endpoints, such as one for each core service, for additional monitoring granularity. For example, a health verification check might check the database, storage, and an external geocoding service that an application uses, with each requiring a different level of uptime and response time. The application could still be healthy if the geocoding service, or some other background task, is unavailable for a few minutes.

Whether to use the same endpoint for monitoring as is used for general access, but to a specific path designed for health verification checks, for example, /health on the general access endpoint. This allows some functional tests in the application to be run by the monitoring tools, such as adding a new user registration, signing in, and placing a test order, while also verifying that the general access endpoint is available.

The type of information to collect in the service in response to monitoring requests, and how to return this information. Most existing tools and frameworks look only at the HTTP status code that the endpoint returns. To return and validate additional information, you might have to create a custom monitoring utility or service.

How much information to collect. Performing excessive processing during the check can overload the application and impact other users. The time it takes might exceed the timeout of the monitoring system so it marks the application as unavailable. Most applications include instrumentation such as error handlers and performance counters that log performance and detailed error information, this might be sufficient instead of returning additional information from a health verification check.

Caching the endpoint status. It could be expensive to run the health check too frequently. If the health status is reported through a dashboard, for example, you don't want every request to the dashboard to trigger a health check. Instead, periodically check the system health and cache the status. Expose an endpoint that returns the cached status.

How to configure security for the monitoring endpoints to protect them from public access, which might expose the application to malicious attacks, risk the exposure of sensitive information, or attract denial of service (DoS) attacks. Typically this should be done in the application configuration so that it can be updated easily without restarting the application. Consider using one or more of the following techniques:

- Secure the endpoint by requiring authentication. You can do this by using an authentication security key in the request header or by passing credentials with the request, provided that the monitoring service or tool supports authentication.
    - Use an obscure or hidden endpoint. For example, expose the endpoint on a different IP address to that used by the default application URL, configure the endpoint on a nonstandard HTTP port, and/or use a complex path to the test page. You can usually specify additional endpoint addresses and ports in the application configuration, and add entries for these endpoints to the DNS server if required to avoid having to specify the IP address directly.
    - Expose a method on an endpoint that accepts a parameter such as a key value or an operation mode value. Depending on the value supplied for this parameter, when a request is received the

code can perform a specific test or set of tests, or return a 404 (Not Found) error if the parameter value isn't recognized. The recognized parameter values could be set in the application configuration.

> DoS attacks are likely to have less impact on a separate endpoint that performs basic functional tests without compromising the operation of the application. Ideally, avoid using a test that might expose sensitive information. If you must return information that might be useful to an attacker, consider how you'll protect the endpoint and the data from unauthorized access. In this case just relying on obscurity isn't enough. You should also consider using an HTTPS connection and encrypting any sensitive data, although this will increase the load on the server.

- How to access an endpoint that's secured using authentication is a point that needs to be considered when evaluating health check endpoints and those that consume it. As an example, App Service's built-in health check integrates with App Service's authentication and authorization features.

How to ensure that the monitoring agent is performing correctly. One approach is to expose an endpoint that simply returns a value from the application configuration or a random value that can be used to test the agent.

> Also ensure that the monitoring system performs checks on itself, such as a self-test and built-in test, to avoid it issuing false positive results.

When to use this pattern

This pattern is useful for:

- Monitoring websites and web applications to verify availability.
- Monitoring websites and web applications to check for correct operation.
- Monitoring middle-tier or shared services to detect and isolate a failure that could disrupt other applications.
- Complementing existing instrumentation in the application, such as performance counters and error handlers. Health verification checking doesn't replace the requirement for logging and auditing in the application. Instrumentation can provide valuable information for an existing framework that monitors counters and error logs to detect failures or other issues. However, it can't provide information if the application is unavailable.

Example

Health Checks for ASP.NET is middleware and a set of libraries for reporting the health of app infrastructure components. It provides a framework for reporting health checks in a consistent method, implementing many of the practices addressed above. This includes external checks like database connectivity and specific concepts like liveness and readiness probes.

A number of example implementations using ASP.NET Health Checks can be found on GitHub.

Monitoring endpoints in Azure hosted applications

Some options for monitoring endpoints in Azure applications are:

- Use the built-in monitoring features of Azure.
- Use a third-party service or a framework such as Microsoft System Center Operations Manager.
- Create a custom utility or a service that runs on your own or on a hosted server.

> Even though Azure provides a reasonably comprehensive set of monitoring options, you can use additional services and tools to provide extra information. Application Insights, a feature of Azure Monitor, is aimed at the development team, to help you understand how your app is performing and how it's being used. It monitors request rates, response times, failure rates, dependency rates and failure rates and it can help you to find out whether external services are slowing you down.

The conditions you can monitor vary depending on the hosting mechanism you choose for your application, but all of these include the ability to create an alert rule that uses a web endpoint you specify in the settings for your service. This endpoint should respond in a timely way so that the alert system can detect that the application is operating correctly.

> Read more information about creating alert notifications.

In the event of a major outage, client traffic should be routable to an application deployment which remain available across other regions or zones. This is ultimately where cross-premises connectivity and global load balancing should be used, depending on whether the application is internal and/or external facing. Services such as Azure Front Door, Azure Traffic Manager, or CDNs can route traffic across regions based on application health provided via health probes.

Azure Traffic Manager is a routing and load-balancing service that can distribute requests to specific instances of your application based on a range of rules and settings. In addition to routing requests, Traffic Manager pings a URL, port, and relative path that you specify on a regular basis to determine which instances of the application defined in its rules are active and are responding to requests. If it detects a status code 200 (OK), it marks the application as available. Any other status code causes Traffic Manager to mark the application as offline. You can view the status in the Traffic Manager console, and configure the rule to reroute requests to other instances of the application that are responding.

However, Traffic Manager will only wait for a certain amount of time to receive a response from the monitoring URL. Therefore, you should ensure that your health verification code executes in this time, allowing for network latency for the round trip from Traffic Manager to your application and back again.

Related guidance

The following guidance can be useful when implementing this pattern:

- Health monitoring Guidance in microservices-based applications
- Well-Architected Framework's Monitoring application health for reliability
- Receiving alert notifications

Index Table pattern

11/19/2021 • 9 minutes to read • Edit Online

Create indexes over the fields in data stores that are frequently referenced by queries. This pattern can improve query performance by allowing applications to more quickly locate the data to retrieve from a data store.

Context and problem

Many data stores organize the data for a collection of entities using the primary key. An application can use this key to locate and retrieve data. The figure shows an example of a data store holding customer information. The primary key is the Customer ID. The figure shows customer information organized by the primary key (Customer ID).

Primary Key (Customer ID)	Customer Data
1	LastName: Smith, Town: Redmond, ...
2	LastName: Jones, Town: Seattle, ...
3	LastName: Robinson, Town: Portland, ...
4	LastName: Brown, Town: Redmond, ...
5	LastName: Smith, Town: Chicago, ...
6	LastName: Green, Town: Redmond, ...
7	LastName: Clarke, Town: Portland, ...
8	LastName: Smith, Town: Redmond, ...
9	LastName: Jones, Town: Chicago, ...
...	...
1000	LastName: Clarke, Town: Chicago, ...
...	...

While the primary key is valuable for queries that fetch data based on the value of this key, an application might not be able to use the primary key if it needs to retrieve data based on some other field. In the customers example, an application can't use the Customer ID primary key to retrieve customers if it queries data solely by referencing the value of some other attribute, such as the town in which the customer is located. To perform a query such as this, the application might have to fetch and examine every customer record, which could be a slow process.

Many relational database management systems support secondary indexes. A secondary index is a separate data structure that's organized by one or more nonprimary (secondary) key fields, and it indicates where the data for each indexed value is stored. The items in a secondary index are typically sorted by the value of the secondary keys to enable fast lookup of data. These indexes are usually maintained automatically by the database management system.

You can create as many secondary indexes as you need to support the different queries that your application performs. For example, in a Customers table in a relational database where the Customer ID is the primary key, it's beneficial to add a secondary index over the town field if the application frequently looks up customers by the town where they reside.

However, although secondary indexes are common in relational systems, most NoSQL data stores used by cloud applications don't provide an equivalent feature.

Solution

If the data store doesn't support secondary indexes, you can emulate them manually by creating your own index tables. An index table organizes the data by a specified key. Three strategies are commonly used for structuring an index table, depending on the number of secondary indexes that are required and the nature of the queries that an application performs.

The first strategy is to duplicate the data in each index table but organize it by different keys (complete denormalization). The next figure shows index tables that organize the same customer information by Town and LastName.

Secondary Key (Town)	Customer Data
Chicago	ID: 5, LastName: Smith, Town: Chicago, ...
Chicago	ID: 9, LastName: Jones, Town: Chicago, ...
Chicago	ID: 1000, LastName: Clarke, Town: Chicago, ...
...	...
Portland	ID: 3, LastName: Robinson, Town: Portland, ...
Portland	ID: 7, LastName: Clarke, Town: Portland, ...
Redmond	ID: 1, LastName: Smith, Town: Redmond, ...
Redmond	ID: 4, LastName: Brown, Town: Redmond, ...
Redmond	ID: 6, LastName: Green, Town: Redmond, ...
Redmond	ID: 8, LastName: Smith, Town: Redmond, ...
Seattle	ID: 2, LastName: Jones, Town: Seattle, ...
...	...

Secondary Key (LastName)	Customer Data
Brown	ID: 4, LastName: Brown, Town: Redmond, ...
Clarke	ID: 7, LastName: Clarke, Town: Portland, ...
Clarke	ID: 1000, LastName: Clarke, Town: Chicago, ...
Green	ID: 6, LastName: Green, Town: Redmond, ...
Jones	ID: 2, LastName: Jones, Town: Seattle, ...
Jones	ID: 9, LastName: Jones, Town: Chicago, ...
...	...
Robinson	ID: 3, LastName: Robinson, Town: Portland, ...
Smith	ID: 1, LastName: Smith, Town: Redmond, ...
Smith	ID: 5, LastName: Smith, Town: Chicago, ...
Smith	ID: 8, LastName: Smith, Town: Redmond, ...
...	...

This strategy is appropriate if the data is relatively static compared to the number of times it's queried using each key. If the data is more dynamic, the processing overhead of maintaining each index table becomes too large for this approach to be useful. Also, if the volume of data is very large, the amount of space required to store the duplicate data is significant.

The second strategy is to create normalized index tables organized by different keys and reference the original data by using the primary key rather than duplicating it, as shown in the following figure. The original data is called a fact table.

This technique saves space and reduces the overhead of maintaining duplicate data. The disadvantage is that an application has to perform two lookup operations to find data using a secondary key. It has to find the primary key for the data in the index table, and then use the primary key to look up the data in the fact table.

The third strategy is to create partially normalized index tables organized by different keys that duplicate frequently retrieved fields. Reference the fact table to access less frequently accessed fields. The next figure shows how commonly accessed data is duplicated in each index table.

With this strategy, you can strike a balance between the first two approaches. The data for common queries can be retrieved quickly by using a single lookup, while the space and maintenance overhead isn't as significant as duplicating the entire data set.

If an application frequently queries data by specifying a combination of values (for example, "Find all customers that live in Redmond and that have a last name of Smith"), you could implement the keys to the items in the index table as a concatenation of the Town attribute and the LastName attribute. The next figure shows an index table based on composite keys. The keys are sorted by Town, and then by LastName for records that have the same value for Town.

Index tables can speed up query operations over sharded data, and are especially useful where the shard key is hashed. The next figure shows an example where the shard key is a hash of the Customer ID. The index table can organize data by the nonhashed value (Town and LastName), and provide the hashed shard key as the lookup data. This can save the application from repeatedly calculating hash keys (an expensive operation) if it needs to retrieve data that falls within a range, or it needs to fetch data in order of the nonhashed key. For example, a query such as "Find all customers that live in Redmond" can be quickly resolved by locating the matching items in the index table, where they're all stored in a contiguous block. Then, follow the references to the customer data using the shard keys stored in the index table.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- The overhead of maintaining secondary indexes can be significant. You must analyze and understand the queries that your application uses. Only create index tables when they're likely to be used regularly. Don't create speculative index tables to support queries that an application doesn't perform, or performs only occasionally.
- Duplicating data in an index table can add significant overhead in storage costs and the effort required to maintain multiple copies of data.
- Implementing an index table as a normalized structure that references the original data requires an application to perform two lookup operations to find data. The first operation searches the index table to retrieve the primary key, and the second uses the primary key to fetch the data.
- If a system incorporates a number of index tables over very large data sets, it can be difficult to maintain consistency between index tables and the original data. It might be possible to design the application around the eventual consistency model. For example, to insert, update, or delete data, an application could post a message to a queue and let a separate task perform the operation and maintain the index tables that reference this data asynchronously. For more information about implementing eventual consistency, see the Data Consistency Primer.

  > Microsoft Azure storage tables support transactional updates for changes made to data held in the same partition (referred to as entity group transactions). If you can store the data for a fact table and one or more index tables in the same partition, you can use this feature to help ensure consistency.

- Index tables might themselves be partitioned or sharded.

When to use this pattern

Use this pattern to improve query performance when an application frequently needs to retrieve data by using a key other than the primary (or shard) key.

This pattern might not be useful when:

- Data is volatile. An index table can become out of date very quickly, making it ineffective or making the overhead of maintaining the index table greater than any savings made by using it.
- A field selected as the secondary key for an index table is nondiscriminating and can only have a small set of

values (for example, gender).

- The balance of the data values for a field selected as the secondary key for an index table are highly skewed. For example, if 90% of the records contain the same value in a field, then creating and maintaining an index table to look up data based on this field might create more overhead than scanning sequentially through the data. However, if queries very frequently target values that lie in the remaining 10%, this index can be useful. You should understand the queries that your application is performing, and how frequently they're performed.

Example

Azure storage tables provide a highly scalable key/value data store for applications running in the cloud. Applications store and retrieve data values by specifying a key. The data values can contain multiple fields, but the structure of a data item is opaque to table storage, which simply handles a data item as an array of bytes.

Azure storage tables also support sharding. The sharding key includes two elements, a partition key and a row key. Items that have the same partition key are stored in the same partition (shard), and the items are stored in row key order within a shard. Table storage is optimized for performing queries that fetch data falling within a contiguous range of row key values within a partition. If you're building cloud applications that store information in Azure tables, you should structure your data with this feature in mind.

For example, consider an application that stores information about movies. The application frequently queries movies by genre (action, documentary, historical, comedy, drama, and so on). You could create an Azure table with partitions for each genre by using the genre as the partition key, and specifying the movie name as the row key, as shown in the next figure.

Partition Key (Genre)	Row Key (Movie Name)	Movie Data
Action	Action Movie 1	Starring Actors: [Fred, Bert], Director: Sid, Date Released 1/1/2013, ...
Action	Action Movie 2	Starring Actors: [Mary, Fred], Director: Harry, Date Released 2/2/2013, ...
Action	Action Movie 3	Starring Actors: [Bill, Ted], Director: Sid, Date Released 3/3/2013, ...
...	...	...

Partition Key (Genre)	Row Key (Movie Name)	Movie Data
Comedy	Comedy Movie 1	Starring Actor: Harry, Director: Sid, Date Released 4/1/2013, ...
Comedy	Comedy Movie 2	Starring Actors: [Alice, Anne], Director: Fred, Date Released 2/1/2013, ...
Comedy	Comedy Movie 3	Starring Actors: [Bert, Bill], Director: Harry, Date Released 3/5/2013, ...
...	...	...

Partition Key (Genre)	Row Key (Movie Name)	Movie Data
Drama	Drama Movie 1	Starring Actor: Keith, Director: Fred, Date Released 1/1/2013, ...
Drama	Drama Movie 2	Starring Actor: Susan, Director: Harry, Date Released 4/5/2013, ...
Drama	Drama Movie 3	Starring Actors: [Keith, Susan], Director: Fred, Date Released 1/8/2013, ...
...	...	...

Shards

This approach is less effective if the application also needs to query movies by starring actor. In this case, you can create a separate Azure table that acts as an index table. The partition key is the actor and the row key is the movie name. The data for each actor will be stored in separate partitions. If a movie stars more than one actor, the same movie will occur in multiple partitions.

You can duplicate the movie data in the values held by each partition by adopting the first approach described in the Solution section above. However, it's likely that each movie will be replicated several times (once for each actor), so it might be more efficient to partially denormalize the data to support the most common queries (such as the names of the other actors) and enable an application to retrieve any remaining details by including the partition key necessary to find the complete information in the genre partitions. This approach is described by the third option in the Solution section. The next figure shows this approach.

Related guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Data Consistency Primer. An index table must be maintained as the data that it indexes changes. In the cloud, it might not be possible or appropriate to perform operations that update an index as part of the same transaction that modifies the data. In that case, an eventually consistent approach is more suitable. Provides information on the issues surrounding eventual consistency.
- Sharding pattern. The Index Table pattern is frequently used in conjunction with data partitioned by using shards. The Sharding pattern provides more information on how to divide a data store into a set of shards.
- Materialized View pattern. Instead of indexing data to support queries that summarize data, it might be more appropriate to create a materialized view of the data. Describes how to support efficient summary queries by generating prepopulated views over data.

Leader Election pattern

11/19/2021 • 11 minutes to read • Edit Online

Coordinate the actions performed by a collection of collaborating instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the others. This can help to ensure that instances don't conflict with each other, cause contention for shared resources, or inadvertently interfere with the work that other instances are performing.

Context and problem

A typical cloud application has many tasks acting in a coordinated manner. These tasks could all be instances running the same code and requiring access to the same resources, or they might be working together in parallel to perform the individual parts of a complex calculation.

The task instances might run separately for much of the time, but it might also be necessary to coordinate the actions of each instance to ensure that they don't conflict, cause contention for shared resources, or accidentally interfere with the work that other task instances are performing.

For example:

- In a cloud-based system that implements horizontal scaling, multiple instances of the same task could be running at the same time with each instance serving a different user. If these instances write to a shared resource, it's necessary to coordinate their actions to prevent each instance from overwriting the changes made by the others.
- If the tasks are performing individual elements of a complex calculation in parallel, the results need to be aggregated when they all complete.

The task instances are all peers, so there isn't a natural leader that can act as the coordinator or aggregator.

Solution

A single task instance should be elected to act as the leader, and this instance should coordinate the actions of the other subordinate task instances. If all of the task instances are running the same code, they are each capable of acting as the leader. Therefore, the election process must be managed carefully to prevent two or more instances taking over the leader role at the same time.

The system must provide a robust mechanism for selecting the leader. This method has to cope with events such as network outages or process failures. In many solutions, the subordinate task instances monitor the leader through some type of heartbeat method, or by polling. If the designated leader terminates unexpectedly, or a network failure makes the leader unavailable to the subordinate task instances, it's necessary for them to elect a new leader.

There are several strategies for electing a leader among a set of tasks in a distributed environment, including:

- Selecting the task instance with the lowest-ranked instance or process ID.
- Racing to acquire a shared, distributed mutex. The first task instance that acquires the mutex is the leader. However, the system must ensure that, if the leader terminates or becomes disconnected from the rest of the system, the mutex is released to allow another task instance to become the leader.
- Implementing one of the common leader election algorithms such as the Bully Algorithm or the Ring Algorithm. These algorithms assume that each candidate in the election has a unique ID, and that it can communicate with the other candidates reliably.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- The process of electing a leader should be resilient to transient and persistent failures.
- It must be possible to detect when the leader has failed or has become otherwise unavailable (such as due to a communications failure). How quickly detection is needed is system dependent. Some systems might be able to function for a short time without a leader, during which a transient fault might be fixed. In other cases, it might be necessary to detect leader failure immediately and trigger a new election.
- In a system that implements horizontal autoscaling, the leader could be terminated if the system scales back and shuts down some of the computing resources.
- Using a shared, distributed mutex introduces a dependency on the external service that provides the mutex. The service constitutes a single point of failure. If it becomes unavailable for any reason, the system won't be able to elect a leader.
- Using a single dedicated process as the leader is a straightforward approach. However, if the process fails there could be a significant delay while it's restarted. The resulting latency can affect the performance and response times of other processes if they're waiting for the leader to coordinate an operation.
- Implementing one of the leader election algorithms manually provides the greatest flexibility for tuning and optimizing the code.

When to use this pattern

Use this pattern when the tasks in a distributed application, such as a cloud-hosted solution, need careful coordination and there's no natural leader.

> Avoid making the leader a bottleneck in the system. The purpose of the leader is to coordinate the work of the subordinate tasks, and it doesn't necessarily have to participate in this work itself—although it should be able to do so if the task isn't elected as the leader.

This pattern might not be useful if:

- There's a natural leader or dedicated process that can always act as the leader. For example, it might be possible to implement a singleton process that coordinates the task instances. If this process fails or becomes unhealthy, the system can shut it down and restart it.
- The coordination between tasks can be achieved using a more lightweight method. For example, if several task instances simply need coordinated access to a shared resource, a better solution is to use optimistic or pessimistic locking to control access.
- A third-party solution is more appropriate. For example, the Microsoft Azure HDInsight service (based on Apache Hadoop) uses the services provided by Apache Zookeeper to coordinate the map and reduce tasks that collect and summarize data.

Example

The DistributedMutex project in the LeaderElection solution (a sample that demonstrates this pattern is available on GitHub) shows how to use a lease on an Azure Storage blob to provide a mechanism for implementing a shared, distributed mutex. This mutex can be used to elect a leader among a group of role instances in an Azure cloud service. The first role instance to acquire the lease is elected the leader, and remains the leader until it releases the lease or isn't able to renew the lease. Other role instances can continue to monitor the blob lease in case the leader is no longer available.

> A blob lease is an exclusive write lock over a blob. A single blob can be the subject of only one lease at any point in time. A role instance can request a lease over a specified blob, and it'll be granted the lease if no other role instance holds a lease over the same blob. Otherwise the request will throw an exception.

> To avoid a faulted role instance retaining the lease indefinitely, specify a lifetime for the lease. When this expires, the lease becomes available. However, while a role instance holds the lease it can request that the lease is renewed, and it'll be granted the lease for a further period of time. The role instance can continually repeat this process if it wants to retain the lease. For more information on how to lease a blob, see Lease Blob (REST API).

The `BlobDistributedMutex` class in the C# example below contains the `RunTaskWhenMutexAcquired` method that enables a role instance to attempt to acquire a lease over a specified blob. The details of the blob (the name, container, and storage account) are passed to the constructor in a `BlobSettings` object when the `BlobDistributedMutex` object is created (this object is a simple struct that is included in the sample code). The constructor also accepts a `Task` that references the code that the role instance should run if it successfully acquires the lease over the blob and is elected the leader. Note that the code that handles the low-level details of acquiring the lease is implemented in a separate helper class named `BlobLeaseManager`.

```
public class BlobDistributedMutex
{
  ...
  private readonly BlobSettings blobSettings;
  private readonly Func<CancellationToken, Task> taskToRunWhenLeaseAcquired;
  ...

  public BlobDistributedMutex(BlobSettings blobSettings,
          Func<CancellationToken, Task> taskToRunWhenLeaseAcquired)
  {
    this.blobSettings = blobSettings;
    this.taskToRunWhenLeaseAcquired = taskToRunWhenLeaseAcquired;
  }

  public async Task RunTaskWhenMutexAcquired(CancellationToken token)
  {
    var leaseManager = new BlobLeaseManager(blobSettings);
    await this.RunTaskWhenBlobLeaseAcquired(leaseManager, token);
  }
  ...
```

The `RunTaskWhenMutexAcquired` method in the code sample above invokes the `RunTaskWhenBlobLeaseAcquired` method shown in the following code sample to actually acquire the lease. The `RunTaskWhenBlobLeaseAcquired` method runs asynchronously. If the lease is successfully acquired, the role instance has been elected the leader. The purpose of the `taskToRunWhenLeaseAcquired` delegate is to perform the work that coordinates the other role instances. If the lease isn't acquired, another role instance has been elected as the leader and the current role instance remains a subordinate. Note that the `TryAcquireLeaseOrWait` method is a helper method that uses the `BlobLeaseManager` object to acquire the lease.

```
  private async Task RunTaskWhenBlobLeaseAcquired(
    BlobLeaseManager leaseManager, CancellationToken token)
  {
    while (!token.IsCancellationRequested)
    {
      // Try to acquire the blob lease.
      // Otherwise wait for a short time before trying again.
      string leaseId = await this.TryAcquireLeaseOrWait(leaseManager, token);

      if (!string.IsNullOrEmpty(leaseId))
      {
        // Create a new linked cancellation token source so that if either the
        // original token is canceled or the lease can't be renewed, the
        // leader task can be canceled.
        using (var leaseCts =
          CancellationTokenSource.CreateLinkedTokenSource(new[] { token }))
        {
          // Run the leader task.
          var leaderTask = this.taskToRunWhenLeaseAcquired.Invoke(leaseCts.Token);
          ...
        }
      }
    }
    ...
  }
```

The task started by the leader also runs asynchronously. While this task is running, the `RunTaskWhenBlobLeaseAcquired` method shown in the following code sample periodically attempts to renew the lease. This helps to ensure that the role instance remains the leader. In the sample solution, the delay between renewal requests is less than the time specified for the duration of the lease in order to prevent another role instance from being elected the leader. If the renewal fails for any reason, the task is canceled.

If the lease fails to be renewed or the task is canceled (possibly as a result of the role instance shutting down), the lease is released. At this point, this or another role instance might be elected as the leader. The code extract below shows this part of the process.

```
  private async Task RunTaskWhenBlobLeaseAcquired(
    BlobLeaseManager leaseManager, CancellationToken token)
  {
    while (...)
    {
      ...
      if (...)
      {
        ...
        using (var leaseCts = ...)
        {
          ...
          // Keep renewing the lease in regular intervals.
          // If the lease can't be renewed, then the task completes.
          var renewLeaseTask =
            this.KeepRenewingLease(leaseManager, leaseId, leaseCts.Token);

          // When any task completes (either the leader task itself or when it
          // couldn't renew the lease) then cancel the other task.
          await CancelAllWhenAnyCompletes(leaderTask, renewLeaseTask, leaseCts);
        }
      }
    }
  }
  ...
}
```

The `KeepRenewingLease` method is another helper method that uses the `BlobLeaseManager` object to renew the lease. The `CancelAllWhenAnyCompletes` method cancels the tasks specified as the first two parameters. The following diagram illustrates using the `BlobDistributedMutex` class to elect a leader and run a task that coordinates operations.

1: A role instance calls the *RunTaskWhenMutexAcquired* method of a *BlobDistibutedMutex* object and is granted the lease over the blob. The role instance is elected the leader.
2: Other role instances call the *RunTaskWhenMutexAcquired* method and are blocked.
3: The *RunTaskWhenMutexAcquired* method in the leader runs a task that coordinates the work of the subordinate role instances.
4: The *RunTaskWhenMutexAcquired* method in the leader periodically renews the lease.

The following code example shows how to use the `BlobDistributedMutex` class in a worker role. This code acquires a lease over a blob named `MyLeaderCoordinatorTask` in the lease's container in development storage, and specifies that the code defined in the `MyLeaderCoordinatorTask` method should run if the role instance is elected the leader.

```
var settings = new BlobSettings(CloudStorageAccount.DevelopmentStorageAccount,
  "leases", "MyLeaderCoordinatorTask");
var cts = new CancellationTokenSource();
var mutex = new BlobDistributedMutex(settings, MyLeaderCoordinatorTask);
mutex.RunTaskWhenMutexAcquired(this.cts.Token);
...

// Method that runs if the role instance is elected the leader
private static async Task MyLeaderCoordinatorTask(CancellationToken token)
{
  ...
}
```

Note the following points about the sample solution:

- The blob is a potential single point of failure. If the blob service becomes unavailable, or is inaccessible, the leader won't be able to renew the lease and no other role instance will be able to acquire the lease. In this case, no role instance will be able to act as the leader. However, the blob service is designed to be resilient, so complete failure of the blob service is considered to be extremely unlikely.
- If the task being performed by the leader stalls, the leader might continue to renew the lease, preventing any other role instance from acquiring the lease and taking over the leader role in order to coordinate tasks. In the real world, the health of the leader should be checked at frequent intervals.
- The election process is nondeterministic. You can't make any assumptions about which role instance will acquire the blob lease and become the leader.
- The blob used as the target of the blob lease shouldn't be used for any other purpose. If a role instance attempts to store data in this blob, this data won't be accessible unless the role instance is the leader and holds the blob lease.

Next steps

The following guidance might also be relevant when implementing this pattern:

- This pattern has a downloadable sample application.
- Autoscaling Guidance. It's possible to start and stop instances of the task hosts as the load on the application varies. Autoscaling can help to maintain throughput and performance during times of peak processing.
- Compute Partitioning Guidance. This guidance describes how to allocate tasks to hosts in a cloud service in a way that helps to minimize running costs while maintaining the scalability, performance, availability, and security of the service.
- The Task-based Asynchronous pattern.
- An example illustrating the Bully Algorithm.
- An example illustrating the Ring Algorithm.
- Apache Curator a client library for Apache ZooKeeper.
- The article Lease Blob (REST API) on MSDN.

Materialized View pattern

11/19/2021 • 7 minutes to read • Edit Online

Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations. This can help support efficient querying and data extraction, and improve application performance.

Context and problem

When storing data, the priority for developers and data administrators is often focused on how the data is stored, as opposed to how it's read. The chosen storage format is usually closely related to the format of the data, requirements for managing data size and data integrity, and the kind of store in use. For example, when using NoSQL document store, the data is often represented as a series of aggregates, each containing all of the information for that entity.

However, this can have a negative effect on queries. When a query only needs a subset of the data from some entities, such as a summary of orders for several customers without all of the order details, it must extract all of the data for the relevant entities in order to obtain the required information.

Solution

To support efficient querying, a common solution is to generate, in advance, a view that materializes the data in a format suited to the required results set. The Materialized View pattern describes generating prepopulated views of data in environments where the source data isn't in a suitable format for querying, where generating a suitable query is difficult, or where query performance is poor due to the nature of the data or the data store.

These materialized views, which only contain data required by a query, allow applications to quickly obtain the information they need. In addition to joining tables or combining data entities, materialized views can include the current values of calculated columns or data items, the results of combining values or executing transformations on the data items, and values specified as part of the query. A materialized view can even be optimized for just a single query.

A key point is that a materialized view and the data it contains is completely disposable because it can be entirely rebuilt from the source data stores. A materialized view is never updated directly by an application, and so it's a specialized cache.

When the source data for the view changes, the view must be updated to include the new information. You can schedule this to happen automatically, or when the system detects a change to the original data. In some cases it might be necessary to regenerate the view manually. The figure shows an example of how the Materialized View pattern might be used.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

How and when the view will be updated. Ideally it'll regenerate in response to an event indicating a change to the source data, although this can lead to excessive overhead if the source data changes rapidly. Alternatively, consider using a scheduled task, an external trigger, or a manual action to regenerate the view.

In some systems, such as when using the Event Sourcing pattern to maintain a store of only the events that modified the data, materialized views are necessary. Prepopulating views by examining all events to determine the current state might be the only way to obtain information from the event store. If you're not using Event Sourcing, you need to consider whether a materialized view is helpful or not. Materialized views tend to be specifically tailored to one, or a small number of queries. If many queries are used, materialized views can result in unacceptable storage capacity requirements and storage cost.

Consider the impact on data consistency when generating the view, and when updating the view if this occurs on a schedule. If the source data is changing at the point when the view is generated, the copy of the data in the view won't be fully consistent with the original data.

Consider where you'll store the view. The view doesn't have to be located in the same store or partition as the original data. It can be a subset from a few different partitions combined.

A view can be rebuilt if lost. Because of that, if the view is transient and is only used to improve query performance by reflecting the current state of the data, or to improve scalability, it can be stored in a cache or in a less reliable location.

When defining a materialized view, maximize its value by adding data items or columns to it based on computation or transformation of existing data items, on values passed in the query, or on combinations of these values when appropriate.

Where the storage mechanism supports it, consider indexing the materialized view to further increase performance. Most relational databases support indexing for views, as do big data solutions based on Apache Hadoop.

When to use this pattern

This pattern is useful when:

- Creating materialized views over data that's difficult to query directly, or where queries must be very complex to extract data that's stored in a normalized, semi-structured, or unstructured way.
- Creating temporary views that can dramatically improve query performance, or can act directly as source

views or data transfer objects for the UI, for reporting, or for display.

- Supporting occasionally connected or disconnected scenarios where connection to the data store isn't always available. The view can be cached locally in this case.
- Simplifying queries and exposing data for experimentation in a way that doesn't require knowledge of the source data format. For example, by joining different tables in one or more databases, or one or more domains in NoSQL stores, and then formatting the data to fit its eventual use.
- Providing access to specific subsets of the source data that, for security or privacy reasons, shouldn't be generally accessible, open to modification, or fully exposed to users.
- Bridging different data stores, to take advantage of their individual capabilities. For example, using a cloud store that's efficient for writing as the reference data store, and a relational database that offers good query and read performance to hold the materialized views.
- When using microservices, you are recommended to keep them loosely coupled, including their data storage. Therefore, materialized views can help you consolidate data from your services. If materialized views are not appropriate in your microservices architecture or specific scenario, please consider having well-defined boundaries that align to domain driven design (DDD) and aggregate their data when requested.

This pattern isn't useful in the following situations:

- The source data is simple and easy to query.
- The source data changes very quickly, or can be accessed without using a view. In these cases, you should avoid the processing overhead of creating views.
- Consistency is a high priority. The views might not always be fully consistent with the original data.

Example

The following figure shows an example of using the Materialized View pattern to generate a summary of sales. Data in the Order, OrderItem, and Customer tables in separate partitions in an Azure storage account are combined to generate a view containing the total sales value for each product in the Electronics category, along with a count of the number of customers who made purchases of each item.

Creating this materialized view requires complex queries. However, by exposing the query result as a materialized view, users can easily obtain the results and use them directly or incorporate them in another query. The view is likely to be used in a reporting system or dashboard, and can be updated on a scheduled basis such as weekly.

> Although this example uses Azure table storage, many relational database management systems also provide native support for materialized views.

Related guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Data Consistency Primer. The summary information in a materialized view has to be maintained so that it reflects the underlying data values. As the data values change, it might not be practical to update the summary data in real time, and instead you'll have to adopt an eventually consistent approach. Summarizes the issues surrounding maintaining consistency over distributed data, and describes the benefits and tradeoffs of different consistency models.
- Command and Query Responsibility Segregation (CQRS) pattern. Use to update the information in a materialized view by responding to events that occur when the underlying data values change.
- Event Sourcing pattern. Use in conjunction with the CQRS pattern to maintain the information in a materialized view. When the data values a materialized view is based on are changed, the system can raise events that describe these changes and save them in an event store.
- Index Table pattern. The data in a materialized view is typically organized by a primary key, but queries might need to retrieve information from this view by examining data in other fields. Use to create secondary indexes over data sets for data stores that don't support native secondary indexes.

Pipes and Filters pattern

11/19/2021 • 12 minutes to read • Edit Online

Decompose a task that performs complex processing into a series of separate elements that can be reused. This can improve performance, scalability, and reusability by allowing task elements that perform the processing to be deployed and scaled independently.

Context and problem

An application is required to perform a variety of tasks of varying complexity on the information that it processes. A straightforward but inflexible approach to implementing an application is to perform this processing as a monolithic module. However, this approach is likely to reduce the opportunities for refactoring the code, optimizing it, or reusing it if parts of the same processing are required elsewhere within the application.

The figure illustrates the issues with processing data using the monolithic approach. An application receives and processes data from two sources. The data from each source is processed by a separate module that performs a series of tasks to transform this data, before passing the result to the business logic of the application.

Some of the tasks that the monolithic modules perform are functionally very similar, but the modules have been designed separately. The code that implements the tasks is closely coupled in a module, and has been developed with little or no thought given to reuse or scalability.

However, the processing tasks performed by each module, or the deployment requirements for each task, could change as business requirements are updated. Some tasks might be compute intensive and could benefit from running on powerful hardware, while others might not require such expensive resources. Also, additional

processing might be required in the future, or the order in which the tasks performed by the processing could change. A solution is required that addresses these issues, and increases the possibilities for code reuse.

Solution

Break down the processing required for each stream into a set of separate components (or filters), each performing a single task. By standardizing the format of the data that each component receives and sends, these filters can be combined together into a pipeline. This helps to avoid duplicating code, and makes it easy to remove, replace, or integrate additional components if the processing requirements change. The next figure shows a solution implemented using pipes and filters.

The time it takes to process a single request depends on the speed of the slowest filter in the pipeline. One or more filters could be a bottleneck, especially if a large number of requests appear in a stream from a particular data source. A key advantage of the pipeline structure is that it provides opportunities for running parallel instances of slow filters, enabling the system to spread the load and improve throughput.

The filters that make up a pipeline can run on different machines, enabling them to be scaled independently and take advantage of the elasticity that many cloud environments provide. A filter that is computationally intensive can run on high-performance hardware, while other less demanding filters can be hosted on less expensive commodity hardware. The filters don't even have to be in the same datacenter or geographic location, which allows each element in a pipeline to run in an environment close to the resources it requires. The next figure shows an example applied to the pipeline for the data from Source 1.

If the input and output of a filter are structured as a stream, it's possible to perform the processing for each filter in parallel. The first filter in the pipeline can start its work and output its results, which are passed directly on to the next filter in the sequence before the first filter has completed its work.

Another benefit is the resiliency that this model can provide. If a filter fails or the machine it's running on is no longer available, the pipeline can reschedule the work that the filter was performing and direct this work to another instance of the component. Failure of a single filter doesn't necessarily result in failure of the entire

pipeline.

Using the Pipes and Filters pattern in conjunction with the Compensating Transaction pattern is an alternative approach to implementing distributed transactions. A distributed transaction can be broken down into separate, compensable tasks, each of which can be implemented by using a filter that also implements the Compensating Transaction pattern. The filters in a pipeline can be implemented as separate hosted tasks running close to the data that they maintain.

Issues and considerations

You should consider the following points when deciding how to implement this pattern:

- **Complexity**. The increased flexibility that this pattern provides can also introduce complexity, especially if the filters in a pipeline are distributed across different servers.
- **Reliability**. Use an infrastructure that ensures that data flowing between filters in a pipeline won't be lost.
- **Idempotency**. If a filter in a pipeline fails after receiving a message and the work is rescheduled to another instance of the filter, part of the work might have already been completed. If this work updates some aspect of the global state (such as information stored in a database), the same update could be repeated. A similar issue might occur if a filter fails after posting its results to the next filter in the pipeline, but before indicating that it's completed its work successfully. In these cases, the same work could be repeated by another instance of the filter, causing the same results to be posted twice. This could result in subsequent filters in the pipeline processing the same data twice. Therefore filters in a pipeline should be designed to be idempotent. For more information, see Idempotency Patterns on Jonathan Oliver's blog.
- **Repeated messages**. If a filter in a pipeline fails after posting a message to the next stage of the pipeline, another instance of the filter might be run, and it'll post a copy of the same message to the pipeline. This could cause two instances of the same message to be passed to the next filter. To avoid this, the pipeline should detect and eliminate duplicate messages.

  > If you're implementing the pipeline by using message queues (such as Microsoft Azure Service Bus queues), the message queuing infrastructure might provide automatic duplicate message detection and removal.

- **Context and state**. In a pipeline, each filter essentially runs in isolation and shouldn't make any assumptions about how it was invoked. This means that each filter should be provided with sufficient context to perform its work. This context could include a large amount of state information.

When to use this pattern

Use this pattern when:

- The processing required by an application can easily be broken down into a set of independent steps.
- The processing steps performed by an application have different scalability requirements.

  > It's possible to group filters that should scale together in the same process. For more information, see the Compute Resource Consolidation pattern.

- Flexibility is required to allow reordering of the processing steps performed by an application, or the capability to add and remove steps.
- The system can benefit from distributing the processing for steps across different servers.

- A reliable solution is required that minimizes the effects of failure in a step while data is being processed.

This pattern might not be useful when:

- The processing steps performed by an application aren't independent, or they have to be performed together as part of the same transaction.
- The amount of context or state information required by a step makes this approach inefficient. It might be possible to persist state information to a database instead, but don't use this strategy if the additional load on the database causes excessive contention.

Example

You can use a sequence of message queues to provide the infrastructure required to implement a pipeline. An initial message queue receives unprocessed messages. A component implemented as a filter task listens for a message on this queue, performs its work, and then posts the transformed message to the next queue in the sequence. Another filter task can listen for messages on this queue, process them, post the results to another queue, and so on until the fully transformed data appears in the final message in the queue. The next figure illustrates implementing a pipeline using message queues.

If you're building a solution on Azure you can use Service Bus queues to provide a reliable and scalable queuing mechanism. The `ServiceBusPipeFilter` class shown below in C# demonstrates how you can implement a filter that receives input messages from a queue, processes these messages, and posts the results to another queue.

> The `ServiceBusPipeFilter` class is defined in the PipesAndFilters.Shared project available from GitHub.

```
public class ServiceBusPipeFilter
{
  ...
  private readonly string inQueuePath;
  private readonly string outQueuePath;
  ...
  private QueueClient inQueue;
  private QueueClient outQueue;
  ...

  public ServiceBusPipeFilter(..., string inQueuePath, string outQueuePath = null)
  {
     ...
     this.inQueuePath = inQueuePath;
     this.outQueuePath = outQueuePath;
  }

  public void Start()
  {
    ...
    // Create the outbound filter queue if it doesn't exist.
    ...
    this.outQueue = QueueClient.CreateFromConnectionString(...);

    ...
    // Create the inbound and outbound queue clients.
    this.inQueue = QueueClient.CreateFromConnectionString(...);
  }
```

```
  public void OnPipeFilterMessageAsync(
    Func<BrokeredMessage, Task<BrokeredMessage>> asyncFilterTask, ...)
  {
    ...

    this.inQueue.OnMessageAsync(
      async (msg) =>
    {
      ...
      // Process the filter and send the output to the
      // next queue in the pipeline.
      var outMessage = await asyncFilterTask(msg);

      // Send the message from the filter processor
      // to the next queue in the pipeline.
      if (outQueue != null)
      {
        await outQueue.SendAsync(outMessage);
      }

      // Note: There's a chance that the same message could be sent twice
      // or that a message gets processed by an upstream or downstream
      // filter at the same time.
      // This would happen in a situation where processing of a message was
      // completed, it was sent to the next pipe/queue, and then failed
      // to complete when using the PeekLock method.
      // Idempotent message processing and concurrency should be considered
      // in a real-world implementation.
    },
    options);
  }

  public async Task Close(TimeSpan timespan)
  {
    // Pause the processing threads.
    this.pauseProcessingEvent.Reset();

    // There's no clean approach for waiting for the threads to complete
    // the processing. This example simply stops any new processing, waits
    // for the existing thread to complete, then closes the message pump
    // and finally returns.
    Thread.Sleep(timespan);

    this.inQueue.Close();
    ...
  }

  ...
}
```

The `Start` method in the `ServiceBusPipeFilter` class connects to a pair of input and output queues, and the `Close` method disconnects from the input queue. The `OnPipeFilterMessageAsync` method performs the actual processing of messages, the `asyncFilterTask` parameter to this method specifies the processing to be performed. The `OnPipeFilterMessageAsync` method waits for incoming messages on the input queue, runs the code specified by the `asyncFilterTask` parameter over each message as it arrives, and posts the results to the output queue. The queues themselves are specified by the constructor.

The sample solution implements filters in a set of worker roles. Each worker role can be scaled independently, depending on the complexity of the business processing that it performs or the resources required for processing. Additionally, multiple instances of each worker role can be run in parallel to improve throughput.

The following code shows an Azure worker role named `PipeFilterARoleEntry`, defined in the PipeFilterA project in the sample solution.

```
public class PipeFilterARoleEntry : RoleEntryPoint
{
  ...
  private ServiceBusPipeFilter pipeFilterA;

  public override bool OnStart()
  {
    ...
    this.pipeFilterA = new ServiceBusPipeFilter(
      ...,
      Constants.QueueAPath,
      Constants.QueueBPath);

    this.pipeFilterA.Start();
    ...
  }

  public override void Run()
  {
    this.pipeFilterA.OnPipeFilterMessageAsync(async (msg) =>
    {
      // Clone the message and update it.
      // Properties set by the broker (Deliver count, enqueue time, ...)
      // aren't cloned and must be copied over if required.
      var newMsg = msg.Clone();

      await Task.Delay(500); // DOING WORK

      Trace.TraceInformation("Filter A processed message:{0} at {1}",
        msg.MessageId, DateTime.UtcNow);

      newMsg.Properties.Add(Constants.FilterAMessageKey, "Complete");

      return newMsg;
    });

    ...
  }

  ...
}
```

This role contains a `ServiceBusPipeFilter` object. The `OnStart` method in the role connects to the queues for receiving input messages and posting output messages (the names of the queues are defined in the `Constants` class). The `Run` method invokes the `OnPipeFilterMessagesAsync` method to perform some processing on each message that's received (in this example, the processing is simulated by waiting for a short period of time). When processing is complete, a new message is constructed containing the results (in this case, the input message has a custom property added), and this message is posted to the output queue.

The sample code contains another worker role named `PipeFilterBRoleEntry` in the PipeFilterB project. This role is similar to `PipeFilterARoleEntry` except that it performs different processing in the `Run` method. In the example solution, these two roles are combined to construct a pipeline, the output queue for the `PipeFilterARoleEntry` role is the input queue for the `PipeFilterBRoleEntry` role.

The sample solution also provides two additional roles named `InitialSenderRoleEntry` (in the InitialSender project) and `FinalReceiverRoleEntry` (in the FinalReceiver project). The `InitialSenderRoleEntry` role provides the initial message in the pipeline. The `OnStart` method connects to a single queue and the `Run` method posts a method to this queue. This queue is the input queue used by the `PipeFilterARoleEntry` role, so posting a message to it causes the message to be received and processed by the `PipeFilterARoleEntry` role. The processed message then passes through the `PipeFilterBRoleEntry` role.

The input queue for the `FinalReceiveRoleEntry` role is the output queue for the `PipeFilterBRoleEntry` role. The `Run` method in the `FinalReceiveRoleEntry` role, shown below, receives the message and performs some final processing. Then it writes the values of the custom properties added by the filters in the pipeline to the trace output.

```
public class FinalReceiverRoleEntry : RoleEntryPoint
{
  ...
  // Final queue/pipe in the pipeline to process data from.
  private ServiceBusPipeFilter queueFinal;

  public override bool OnStart()
  {
    ...
    // Set up the queue.
    this.queueFinal = new ServiceBusPipeFilter(...,Constants.QueueFinalPath);
    this.queueFinal.Start();
    ...
  }

  public override void Run()
  {
    this.queueFinal.OnPipeFilterMessageAsync(
      async (msg) =>
      {
        await Task.Delay(500); // DOING WORK

        // The pipeline message was received.
        Trace.TraceInformation(
          "Pipeline Message Complete - FilterA:{0} FilterB:{1}",
          msg.Properties[Constants.FilterAMessageKey],
          msg.Properties[Constants.FilterBMessageKey]);

        return null;
      });
    ...
  }

  ...
}
```

Next steps

The following guidance might also be relevant when implementing this pattern:

- A sample that demonstrates this pattern is available on GitHub.

Related guidance

The following patterns might also be relevant when implementing this pattern:

- Competing Consumers pattern. A pipeline can contain multiple instances of one or more filters. This approach is useful for running parallel instances of slow filters, enabling the system to spread the load and improve throughput. Each instance of a filter will compete for input with the other instances, two instances of a filter shouldn't be able to process the same data. Provides an explanation of this approach.
- Compute Resource Consolidation pattern. It might be possible to group filters that should scale together into the same process. Provides more information about the benefits and tradeoffs of this strategy.
- Compensating Transaction pattern. A filter can be implemented as an operation that can be reversed, or that has a compensating operation that restores the state to a previous version in the event of a failure. Explains how this can be implemented to maintain or achieve eventual consistency.

- Idempotency patterns on Jonathan Oliver's blog.

Priority Queue pattern

11/19/2021 • 9 minutes to read • Edit Online

Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority. This pattern is useful in applications that offer different service level guarantees to individual clients.

Context and problem

Applications can delegate specific tasks to other services, for example, to perform background processing or to integrate with other applications or services. In the cloud, a message queue is typically used to delegate tasks to background processing. In many cases the order requests are received in by a service isn't important. In some cases, though, it's necessary to prioritize specific requests. These requests should be processed earlier than lower priority requests that were sent previously by the application.

Solution

A queue is usually a first-in, first-out (FIFO) structure, and consumers typically receive messages in the same order that they were posted to the queue. However, some message queues support priority messaging. The application posting a message can assign a priority and the messages in the queue are automatically reordered so that those with a higher priority will be received before those with a lower priority. The figure illustrates a queue with priority messaging.

> Most message queue implementations support multiple consumers (following the Competing Consumers pattern), and the number of consumer processes can be scaled up or down depending on demand.

In systems that don't support priority-based message queues, an alternative solution is to maintain a separate queue for each priority. The application is responsible for posting messages to the appropriate queue. Each queue can have a separate pool of consumers. Higher priority queues can have a larger pool of consumers running on faster hardware than lower priority queues. The next figure illustrates using separate message queues for each priority.

A variation on this strategy is to have a single pool of consumers that check for messages on high priority queues first, and only then start to fetch messages from lower priority queues. There are some semantic differences between a solution that uses a single pool of consumer processes (either with a single queue that supports messages with different priorities or with multiple queues that each handle messages of a single priority), and a solution that uses multiple queues with a separate pool for each queue.

In the single pool approach, higher priority messages are always received and processed before lower priority messages. In theory, messages that have a very low priority could be continually superseded and might never be processed. In the multiple pool approach, lower priority messages will always be processed, just not as quickly as those of a higher priority (depending on the relative size of the pools and the resources that they have available).

Using a priority queuing mechanism can provide the following advantages:

- It allows applications to meet business requirements that require prioritization of availability or performance, such as offering different levels of service to specific groups of customers.
- It can help to minimize operational costs. In the single queue approach, you can scale back the number of consumers if necessary. High priority messages will still be processed first (although possibly more slowly), and lower priority messages might be delayed for longer. If you've implemented the multiple message queue approach with separate pools of consumers for each queue, you can reduce the pool of consumers for lower priority queues, or even suspend processing for some very low priority queues by stopping all the consumers that listen for messages on those queues.
- The multiple message queue approach can help maximize application performance and scalability by partitioning messages based on processing requirements. For example, vital tasks can be prioritized to be handled by receivers that run immediately while less important background tasks can be handled by receivers that are scheduled to run at less busy periods.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

Define the priorities in the context of the solution. For example, high priority could mean that messages should be processed within ten seconds. Identify the requirements for handling high priority items, and the other resources that should be allocated to meet these criteria.

Decide if all high priority items must be processed before any lower priority items. If the messages are being processed by a single pool of consumers, you have to provide a mechanism that can preempt and suspend a task that's handling a low priority message if a higher priority message becomes available.

In the multiple queue approach, when using a single pool of consumer processes that listen on all queues rather than a dedicated consumer pool for each queue, the consumer must apply an algorithm that ensures it always services messages from higher priority queues before those from lower priority queues.

Monitor the processing speed on high and low priority queues to ensure that messages in these queues are processed at the expected rates.

If you need to guarantee that low priority messages will be processed, it's necessary to implement the multiple message queue approach with multiple pools of consumers. Alternatively, in a queue that supports message prioritization, it's possible to dynamically increase the priority of a queued message as it ages. However, this approach depends on the message queue providing this feature.

Using a separate queue for each message priority works best for systems that have a small number of well-defined priorities.

Message priorities can be determined logically by the system. For example, rather than having explicit high and low priority messages, they could be designated as "fee paying customer," or "non-fee paying customer." Depending on your business model, your system can allocate more resources to processing messages from fee paying customers than non-fee paying ones.

There might be a financial and processing cost associated with checking a queue for a message (some commercial messaging systems charge a small fee each time a message is posted or retrieved, and each time a queue is queried for messages). This cost increases when checking multiple queues.

It's possible to dynamically adjust the size of a pool of consumers based on the length of the queue that the pool is servicing. For more information, see the Autoscaling Guidance.

When to use this pattern

This pattern is useful in scenarios where:

- The system must handle multiple tasks that have different priorities.
- Different users or tenants should be served with different priority.

Example

Microsoft Azure doesn't provide a queuing mechanism that natively supports automatic prioritization of messages through sorting. However, it does provide Azure Service Bus topics and subscriptions that support a queuing mechanism that provides message filtering, together with a wide range of flexible capabilities that make it ideal for use in most priority queue implementations.

An Azure solution can implement a Service Bus topic an application can post messages to, in the same way as a queue. Messages can contain metadata in the form of application-defined custom properties. Service Bus subscriptions can be associated with the topic, and these subscriptions can filter messages based on their properties. When an application sends a message to a topic, the message is directed to the appropriate subscription where it can be read by a consumer. Consumer processes can retrieve messages from a subscription using the same semantics as a message queue (a subscription is a logical queue). The following figure illustrates implementing a priority queue with Azure Service Bus topics and subscriptions.

In the figure above, the application creates several messages and assigns a custom property called `Priority` in each message with a value, either `High` or `Low`. The application posts these messages to a topic. The topic has two associated subscriptions that both filter messages by examining the `Priority` property. One subscription accepts messages where the `Priority` property is set to `High`, and the other accepts messages where the `Priority` property is set to `Low`. A pool of consumers reads messages from each subscription. The high priority subscription has a larger pool, and these consumers might be running on more powerful computers with more resources available than the consumers in the low priority pool.

Note that there's nothing special about the designation of high and low priority messages in this example. They're simply labels specified as properties in each message, and are used to direct messages to a specific subscription. If additional priorities are required, it's relatively easy to create further subscriptions and pools of consumer processes to handle these priorities.

The PriorityQueue solution available on GitHub contains an implementation of this approach. This solution contains two worker role projects named `PriorityQueue.High` and `PriorityQueue.Low`. These worker roles inherit from the `PriorityWorkerRole` class that contains the functionality for connecting to a specified subscription in the `OnStart` method.

The `PriorityQueue.High` and `PriorityQueue.Low` worker roles connect to different subscriptions, defined by their configuration settings. An administrator can configure different numbers of each role to be run. Typically there'll be more instances of the `PriorityQueue.High` worker role than the `PriorityQueue.Low` worker role.

The `Run` method in the `PriorityWorkerRole` class arranges for the virtual `ProcessMessage` method (also defined in the `PriorityWorkerRole` class) to be run for each message received on the queue. The following code shows the `Run` and `ProcessMessage` methods. The `QueueManager` class, defined in the PriorityQueue.Shared project, provides helper methods for using Azure Service Bus queues.

```
public class PriorityWorkerRole : RoleEntryPoint
{
  private QueueManager queueManager;
  ...

  public override void Run()
  {
    // Start listening for messages on the subscription.
    var subscriptionName = CloudConfigurationManager.GetSetting("SubscriptionName");
    this.queueManager.ReceiveMessages(subscriptionName, this.ProcessMessage);
    ...;
  }
  ...

  protected virtual async Task ProcessMessage(BrokeredMessage message)
  {
    // Simulating processing.
    await Task.Delay(TimeSpan.FromSeconds(2));
  }
}
```

The `PriorityQueue.High` and `PriorityQueue.Low` worker roles both override the default functionality of the `ProcessMessage` method. The code below shows the `ProcessMessage` method for the `PriorityQueue.High` worker role.

```
protected override async Task ProcessMessage(BrokeredMessage message)
{
  // Simulate message processing for High priority messages.
  await base.ProcessMessage(message);
  Trace.TraceInformation("High priority message processed by " +
    RoleEnvironment.CurrentRoleInstance.Id + " MessageId: " + message.MessageId);
}
```

When an application posts messages to the topic associated with the subscriptions used by the `PriorityQueue.High` and `PriorityQueue.Low` worker roles, it specifies the priority by using the `Priority` custom property, as shown in the following code example. This code (implemented in the `WorkerRole` class in the PriorityQueue.Sender project), uses the `SendBatchAsync` helper method of the `QueueManager` class to post messages to a topic in batches.

```
// Send a low priority batch.
var lowMessages = new List<BrokeredMessage>();

for (int i = 0; i < 10; i++)
{
  var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
  message.Properties["Priority"] = Priority.Low;
  lowMessages.Add(message);
}

this.queueManager.SendBatchAsync(lowMessages).Wait();
...

// Send a high priority batch.
var highMessages = new List<BrokeredMessage>();

for (int i = 0; i < 10; i++)
{
  var message = new BrokeredMessage() { MessageId = Guid.NewGuid().ToString() };
  message.Properties["Priority"] = Priority.High;
  highMessages.Add(message);
}

this.queueManager.SendBatchAsync(highMessages).Wait();
```

Next steps

The following guidance might also be relevant when implementing this pattern:

- A sample that demonstrates this pattern is available on GitHub.
- Asynchronous Messaging Primer. A consumer service that processes a request might need to send a reply to the instance of the application that posted the request. Provides information on the strategies that you can use to implement request/response messaging.
- Autoscaling Guidance. It might be possible to scale the size of the pool of consumer processes handling a queue depending on the length of the queue. This strategy can help to improve performance, especially for pools handling high priority messages.

Related patterns and guidance

The following patterns might also be relevant when implementing this pattern:

- Competing Consumers pattern. To increase the throughput of the queues, it's possible to have multiple consumers that listen on the same queue, and process the tasks in parallel. These consumers will compete for messages, but only one should be able to process each message. Provides more information on the benefits and tradeoffs of implementing this approach.
- Throttling pattern. You can implement throttling by using queues. Priority messaging can be used to ensure that requests from critical applications, or applications being run by high-value customers, are given priority over requests from less important applications.
- Enterprise Integration patterns with Service Bus on Abhishek Lal's blog.

Publisher-Subscriber pattern

11/19/2021 • 6 minutes to read • Edit Online

Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.

Also called: Pub/sub messaging

Context and problem

In cloud-based and distributed applications, components of the system often need to provide information to other components as events happen.

Asynchronous messaging is an effective way to decouple senders from consumers, and avoid blocking the sender to wait for a response. However, using a dedicated message queue for each consumer does not effectively scale to many consumers. Also, some of the consumers might be interested in only a subset of the information. How can the sender announce events to all interested consumers without knowing their identities?

Solution

Introduce an asynchronous messaging subsystem that includes the following:

- An input messaging channel used by the sender. The sender packages events into messages, using a known message format, and sends these messages via the input channel. The sender in this pattern is also called the *publisher*.

  > **NOTE**
  >
  > A *message* is a packet of data. An *event* is a message that notifies other components about a change or an action that has taken place.

- One output messaging channel per consumer. The consumers are known as *subscribers*.
- A mechanism for copying each message from the input channel to the output channels for all subscribers interested in that message. This operation is typically handled by an intermediary such as a message broker or event bus.

The following diagram shows the logical components of this pattern:

Pub/sub messaging has the following benefits:

- It decouples subsystems that still need to communicate. Subsystems can be managed independently, and messages can be properly managed even if one or more receivers are offline.
- It increases scalability and improves responsiveness of the sender. The sender can quickly send a single message to the input channel, then return to its core processing responsibilities. The messaging infrastructure is responsible for ensuring messages are delivered to interested subscribers.
- It improves reliability. Asynchronous messaging helps applications continue to run smoothly under increased loads and handle intermittent failures more effectively.
- It allows for deferred or scheduled processing. Subscribers can wait to pick up messages until off-peak hours, or messages can be routed or processed according to a specific schedule.
- It enables simpler integration between systems using different platforms, programming languages, or communication protocols, as well as between on-premises systems and applications running in the cloud.
- It facilitates asynchronous workflows across an enterprise.
- It improves testability. Channels can be monitored and messages can be inspected or logged as part of an overall integration test strategy.
- It provides separation of concerns for your applications. Each application can focus on its core capabilities, while the messaging infrastructure handles everything required to reliably route messages to multiple consumers.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- **Existing technologies.** It is strongly recommended to use available messaging products and services that support a publish-subscribe model, rather than building your own. In Azure, consider using Service Bus, Event Hubs or Event Grid. Other technologies that can be used for pub/sub messaging include Redis, RabbitMQ, and Apache Kafka.
- **Subscription handling.** The messaging infrastructure must provide mechanisms that consumers can use to subscribe to or unsubscribe from available channels.
- **Security.** Connecting to any message channel must be restricted by security policy to prevent eavesdropping by unauthorized users or applications.
- **Subsets of messages.** Subscribers are usually only interested in subset of the messages distributed by a publisher. Messaging services often allow subscribers to narrow the set of messages received by:
  - **Topics.** Each topic has a dedicated output channel, and each consumer can subscribe to all relevant topics.
  - **Content filtering.** Messages are inspected and distributed based on the content of each message. Each subscriber can specify the content it is interested in.
- **Wildcard subscribers.** Consider allowing subscribers to subscribe to multiple topics via wildcards.
- **Bi-directional communication.** The channels in a publish-subscribe system are treated as unidirectional. If a specific subscriber needs to send acknowledgment or communicate status back to the publisher, consider using the Request/Reply Pattern. This pattern uses one channel to send a message to the subscriber, and a separate reply channel for communicating back to the publisher.
- **Message ordering.** The order in which consumer instances receive messages isn't guaranteed, and doesn't necessarily reflect the order in which the messages were created. Design the system to ensure that message processing is idempotent to help eliminate any dependency on the order of message

handling.

- **Message priority.** Some solutions may require that messages are processed in a specific order. The Priority Queue pattern provides a mechanism for ensuring specific messages are delivered before others.
- **Poison messages.** A malformed message, or a task that requires access to resources that aren't available, can cause a service instance to fail. The system should prevent such messages being returned to the queue. Instead, capture and store the details of these messages elsewhere so that they can be analyzed if necessary.
- **Repeated messages.** The same message might be sent more than once. For example, the sender might fail after posting a message. Then a new instance of the sender might start up and repeat the message. The messaging infrastructure should implement duplicate message detection and removal (also known as de-duping) based on message IDs in order to provide at-most-once delivery of messages.
- **Message expiration.** A message might have a limited lifetime. If it isn't processed within this period, it might no longer be relevant and should be discarded. A sender can specify an expiration time as part of the data in the message. A receiver can examine this information before deciding whether to perform the business logic associated with the message.
- **Message scheduling.** A message might be temporarily embargoed and should not be processed until a specific date and time. The message should not be available to a receiver until this time.

When to use this pattern

Use this pattern when:

- An application needs to broadcast information to a significant number of consumers.
- An application needs to communicate with one or more independently-developed applications or services, which may use different platforms, programming languages, and communication protocols.
- An application can send information to consumers without requiring real-time responses from the consumers.
- The systems being integrated are designed to support an eventual consistency model for their data.
- An application needs to communicate information to multiple consumers, which may have different availability requirements or uptime schedules than the sender.

This pattern might not be useful when:

- An application has only a few consumers who need significantly different information from the producing application.
- An application requires near real-time interaction with consumers.

Example

The following diagram shows an enterprise integration architecture that uses Service Bus to coordinate workflows, and Event Grid to notify subsystems of events that occur. For more information, see Enterprise integration on Azure using message queues and events.

Next steps

The following guidance might be relevant when implementing this pattern:

- Choose between Azure services that deliver messages.
- The Event-driven architecture style is an architecture style that uses pub/sub messaging.
- Asynchronous Messaging Primer. Message queues are an asynchronous communications mechanism. If a consumer service needs to send a reply to an application, it might be necessary to implement some form of response messaging. The Asynchronous Messaging Primer provides information on how to implement request/reply messaging using message queues.

Related guidance

The following patterns might be relevant when implementing this pattern:

- Observer pattern. The Publish-Subscribe pattern builds on the Observer pattern by decoupling subjects from observers via asynchronous messaging.
- Message Broker pattern. Many messaging subsystems that support a publish-subscribe model are implemented via a message broker.

Queue-Based Load Leveling pattern

11/19/2021 • 5 minutes to read • Edit Online

Use a queue that acts as a buffer between a task and a service it invokes in order to smooth intermittent heavy loads that can cause the service to fail or the task to time out. This can help to minimize the impact of peaks in demand on availability and responsiveness for both the task and the service.

Context and problem

Many solutions in the cloud involve running tasks that invoke services. In this environment, if a service is subjected to intermittent heavy loads, it can cause performance or reliability issues.

A service could be part of the same solution as the tasks that use it, or it could be a third-party service providing access to frequently used resources such as a cache or a storage service. If the same service is used by a number of tasks running concurrently, it can be difficult to predict the volume of requests to the service at any time.

A service might experience peaks in demand that cause it to overload and be unable to respond to requests in a timely manner. Flooding a service with a large number of concurrent requests can also result in the service failing if it's unable to handle the contention these requests cause.

Solution

Refactor the solution and introduce a queue between the task and the service. The task and the service run asynchronously. The task posts a message containing the data required by the service to a queue. The queue acts as a buffer, storing the message until it's retrieved by the service. The service retrieves the messages from the queue and processes them. Requests from a number of tasks, which can be generated at a highly variable rate, can be passed to the service through the same message queue. This figure shows using a queue to level the load on a service.

The queue decouples the tasks from the service, and the service can handle the messages at its own pace regardless of the volume of requests from concurrent tasks. Additionally, there's no delay to a task if the service isn't available at the time it posts a message to the queue.

This pattern provides the following benefits:

- It can help to maximize availability because delays arising in services won't have an immediate and direct impact on the application, which can continue to post messages to the queue even when the service isn't available or isn't currently processing messages.
- It can help to maximize scalability because both the number of queues and the number of services can be

varied to meet demand.

- It can help to control costs because the number of service instances deployed only have to be adequate to meet average load rather than the peak load.

> Some services implement throttling when demand reaches a threshold beyond which the system could fail. Throttling can reduce the functionality available. You can implement load leveling with these services to ensure that this threshold isn't reached.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- It's necessary to implement application logic that controls the rate at which services handle messages to avoid overwhelming the target resource. Avoid passing spikes in demand to the next stage of the system. Test the system under load to ensure that it provides the required leveling, and adjust the number of queues and the number of service instances that handle messages to achieve this.
- Message queues are a one-way communication mechanism. If a task expects a reply from a service, it might be necessary to implement a mechanism that the service can use to send a response. For more information, see the Asynchronous Messaging Primer.
- Be careful if you apply autoscaling to services that are listening for requests on the queue. This can result in increased contention for any resources that these services share and diminish the effectiveness of using the queue to level the load.

When to use this pattern

This pattern is useful to any application that uses services that are subject to overloading.

This pattern isn't useful if the application expects a response from the service with minimal latency.

Example

A web app writes data to an external data store. If a large number of instances of the web app run concurrently, the data store might be unable to respond to requests quickly enough, causing requests to time out, be throttled, or otherwise fail. The following diagram shows a data store being overwhelmed by a large number of concurrent requests from instances of an application.

To resolve this, you can use a queue to level the load between the application instances and the data store. An Azure Functions app reads the messages from the queue and performs the read/write requests to the data store. The application logic in the function app can control the rate at which it passes requests to the data store, to prevent the store from being overwhelmed. (Otherwise the function app will just re-introduce the same problem at the back end.)

Next steps

The following guidance might also be relevant when implementing this pattern:

- Asynchronous Messaging Primer. Message queues are inherently asynchronous. It might be necessary to redesign the application logic in a task if it's adapted from communicating directly with a service to using

a message queue. Similarly, it might be necessary to refactor a service to accept requests from a message queue. Alternatively, it might be possible to implement a proxy service, as described in the example.

- Choose between Azure messaging services. Information about choosing a messaging and queuing mechanism in Azure applications.
- Improve scalability in an Azure web application. This reference architecture includes queue-based load leveling as part of the architecture.

Related guidance

The following patterns might also be relevant when implementing this pattern:

- Competing Consumers pattern. It might be possible to run multiple instances of a service, each acting as a message consumer from the load-leveling queue. You can use this approach to adjust the rate at which messages are received and passed to a service.
- Throttling pattern. A simple way to implement throttling with a service is to use queue-based load leveling and route all requests to a service through a message queue. The service can process requests at a rate that ensures that resources required by the service aren't exhausted, and to reduce the amount of contention that could occur.

Rate Limiting pattern

11/19/2021 • 10 minutes to read • Edit Online

Many services use a throttling pattern to control the resources they consume, imposing limits on the rate at which other applications or services can access them. You can use a rate limiting pattern to help you avoid or minimize throttling errors related to these throttling limits and to help you more accurately predict throughput.

A rate limiting pattern is appropriate in many scenarios, but it is particularly helpful for large-scale repetitive automated tasks such as batch processing.

Context and problem

Performing large numbers of operations using a throttled service can result in increased traffic and throughput, as you'll need to both track rejected requests and then retry those operations. As the number of operations increases, a throttling limit may require multiple passes of resending data, resulting in a larger performance impact.

As an example, consider the following naive retry on error process for ingesting data into Cosmos DB:

1. Your application needs to ingest 10,000 records into Cosmos DB. Each record costs 10 Request Units (RUs) to ingest, requiring a total of 100,000 RUs to complete the job.
2. Your Cosmos DB instance has 20,000 RUs provisioned capacity.
3. You send all 10,000 records to Cosmos DB. 2,000 records are written successfully and 8,000 records are rejected.
4. You send the remaining 8,000 records to Cosmos DB. 2,000 records are written successfully and 6,000 records are rejected.
5. You send the remaining 6,000 records to Cosmos DB. 2,000 records are written successfully and 4,000 records are rejected.
6. You send the remaining 4,000 records to Cosmos DB. 2,000 records are written successfully and 2,000 records are rejected.
7. You send the remaining 2,000 records to Cosmos DB. All are written successfully.

The ingestion job completed successfully, but only after sending 30,000 records to Cosmos DB even though the entire data set only consisted of 10,000 records.

There are additional factors to consider in the above example:

- Large numbers of errors can also result in additional work to log these errors and process the resulting log data. This naive approach will have handled 20,000 errors, and logging these errors may impose a processing, memory, or storage resource cost.
- Not knowing the throttling limits of the ingestion service, the naive approach has no way to set expectations for how long data processing will take. Rate limiting can allow you to calculate the time required for ingestion.

Solution

Rate limiting can reduce your traffic and potentially improve throughput by reducing the number of records sent to a service over a given period of time.

A service may throttle based on different metrics over time, such as:

- The number of operations (for example, 20 requests per second).
- The amount of data (for example, 2 GiB per minute).
- The relative cost of operations (for example, 20,000 RUs per second).

Regardless of the metric used for throttling, your rate limiting implementation will involve controlling the number and/or size of operations sent to the service over a specific time period, optimizing your use of the service while not exceeding its throttling capacity.

In scenarios where your APIs can handle requests faster than any throttled ingestion services allow, you'll need to manage how quickly you can use the service. However, only treating the throttling as a data rate mismatch problem, and simply buffering your ingestion requests until the throttled service can catch up, is risky. If your application crashes in this scenario, you risk losing any of this buffered data.

To avoid this risk, consider sending your records to a durable messaging system that *can* handle your full ingestion rate. (Services such as Azure Event Hubs can handle millions of operations per second). You can then use one or more job processors to read the records from the messaging system at a controlled rate that is within the throttled service's limits. Submitting records to the messaging system can save internal memory by allowing you to dequeue only the records that can be processed during a given time interval.

Azure provides several durable messaging services that you can use with this pattern, including:

- Azure Service Bus
- Azure Queue Storage
- Azure Event Hubs

When you're sending records, the time period you use for releasing records may be more granular than the period the service throttles on. Systems often set throttles based on timespans you can easily comprehend and work with. However, for the computer running a service, these timeframes may be very long compared to how fast it can process information. For instance, a system might throttle per second or per minute, but commonly the code is processing on the order of nanoseconds or milliseconds.

While not required, it's often recommended to send smaller amounts of records more frequently to improve throughput. So rather than trying to batch things up for a release once a second or once a minute, you can be more granular than that to keep your resource consumption (memory, CPU, network, etc.) flowing at a more even rate, preventing potential bottlenecks due to sudden bursts of requests. For example, if a service allows 100 operations per second, the implementation of a rate limiter may even out requests by releasing 20 operations every 200 milliseconds, as shown in the following graph.

In addition, it's sometimes necessary for multiple uncoordinated processes to share a throttled service. To implement rate limiting in this scenario you can logically partition the service's capacity and then use a distributed mutual exclusion system to manage exclusive locks on those partitions. The uncoordinated processes can then compete for locks on those partitions whenever they need capacity. For each partition that a process holds a lock for, it's granted a certain amount of capacity.

For example, if the throttled system allows 500 requests per second, you might create 20 partitions worth 25 requests per second each. If a process needed to issue 100 requests, it might ask the distributed mutual exclusion system for four partitions. The system might grant two partitions for 10 seconds. The process would then rate limit to 50 requests per second, complete the task in two seconds, and then release the lock.

One way to implement this pattern would be to use Azure Storage. In this scenario, you create one 0-byte blob per logical partition in a container. Your applications can then obtain exclusive leases directly against those blobs for a short period of time (for example, 15 seconds). For every lease an application is granted, it will be able to use that partition's worth of capacity. The application then needs to track the lease time so that, when it expires, it can stop using the capacity it was granted. When implementing this pattern, you'll often want each process to attempt to lease a random partition when it needs capacity.

To further reduce latency, you might allocate a small amount of exclusive capacity for each process. A process would then only seek to obtain a lease on shared capacity if it needed to exceed its reserved capacity.

As an alternative to Azure Storage, you could also implement this kind of lease management system using technologies such as Zookeeper, Consul, etcd, Redis/Redsync, and others.

Issues and considerations

Consider the following when deciding how to implement this pattern:

- While the rate limiting pattern can reduce the number of throttling errors, your application will still need to properly handle any throttling errors that may occur.
- If your application has multiple workstreams that access the same throttled service, you'll need to integrate all of them into your rate limiting strategy. For instance, you might support bulk loading records into a database but also querying for records in that same database. You can manage capacity by ensuring all workstreams are gated through the same rate limiting mechanism. Alternatively, you might reserve separate pools of capacity for each workstream.
- The throttled service may be used in multiple applications. In some—but not all—cases it is possible to coordinate that usage (as shown above). If you start seeing a larger than expected number of throttling errors, this may be a sign of contention between applications accessing a service. If so, you may need to consider temporarily reducing the throughput imposed by your rate limiting mechanism until the usage from other applications lowers.

When to use this pattern

Use this pattern to:

- Reduce throttling errors raised by a throttle-limited service.
- Reduce traffic compared to a naive retry on error approach.
- Reduce memory consumption by dequeuing records only when there is capacity to process them.

Example

The following example application allows users to submit records of various types to an API. There is a unique job processor for each record type that performs the following steps:

1. Validation
2. Enrichment
3. Insertion of the record into the database

All components of the application (API, job processor A, and job processor B) are separate processes that may be scaled independently. The processes do not directly communicate with one another.

This diagram incorporates the following workflow:

1. A user submits 10,000 records of type A to the API.

2. The API enqueues those 10,000 records in Queue A.
3. A user submits 5,000 records of type B to the API.
4. The API enqueues those 5,000 records in Queue B.
5. Job Processor A sees Queue A has records and tries to gain an exclusive lease on blob 2.
6. Job Processor B sees Queue B has records and tries to gain an exclusive lease on blob 2.
7. Job Processor A fails to obtain the lease.
8. Job Processor B obtains the lease on blob 2 for 15 seconds. It can now rate limit requests to the database at a rate of 100 per second.
9. Job Processor B dequeues 100 records from Queue B and writes them.
10. One second passes.
11. Job Processor A sees Queue A has more records and tries to gain an exclusive lease on blob 6.
12. Job Processor B sees Queue B has more records and tries to gain an exclusive lease on blob 3.
13. Job Processor A obtains the lease on blob 6 for 15 seconds. It can now rate limit requests to the database at a rate of 100 per second.
14. Job Processor B obtains the lease on blob 3 for 15 seconds. It can now rate limit requests to the database at a rate of 200 per second. (It also holds the lease for blob 2.)
15. Job Processor A dequeues 100 records from Queue A and writes them.
16. Job Processor B dequeues 200 records from Queue B and writes them.
17. One second passes.
18. Job Processor A sees Queue A has more records and tries to gain an exclusive lease on blob 0.
19. Job Processor B sees Queue B has more records and tries to gain an exclusive lease on blob 1.
20. Job Processor A obtains the lease on blob 0 for 15 seconds. It can now rate limit requests to the database at a rate of 200 per second. (It also holds the lease for blob 6.)
21. Job Processor B obtains the lease on blob 1 for 15 seconds. It can now rate limit requests to the database at a rate of 300 per second. (It also holds the lease for blobs 2 and 3.)
22. Job Processor A dequeues 200 records from Queue A and writes them.
23. Job Processor B dequeues 300 records from Queue B and writes them.
24. And so on...

After 15 seconds, one or both jobs still will not be completed. As the leases expire, a processor should also reduce the number of requests it dequeues and writes.

An implementation of this pattern, implemented in Go, is available on GitHub.

Related guidance

The following patterns and guidance might also be relevant when implementing this pattern:

- Throttling. The rate limiting pattern discussed here is typically implemented in response to a service that is throttled.
- Retry. When requests to throttled service result in throttling errors, it's generally appropriate to retry those after an appropriate interval.

Queue-Based Load Leveling is similar but differs from the Rate Limiting pattern in several key ways:

1. Rate limiting doesn't necessarily need to use queues to manage load, but it does need to make use of a durable messaging service. For example, a rate limiting pattern can make use of services like Apache Kafka or Azure Event Hubs.
2. The rate limiting pattern introduces the concept of a distributed mutual exclusion system on partitions, which allows you to manage capacity for multiple uncoordinated processes that communicate with the same throttled service.

3. A queue-based load leveling pattern is applicable anytime there is a performance mismatch between services or to improve resilience. This makes it a broader pattern than rate limiting, which is more specifically concerned with efficiently accessing a throttled service.

Retry pattern

11/19/2021 • 10 minutes to read • Edit Online

Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation. This can improve the stability of the application.

Context and problem

An application that communicates with elements running in the cloud has to be sensitive to the transient faults that can occur in this environment. Faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that occur when a service is busy.

These faults are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay it's likely to be successful. For example, a database service that's processing a large number of concurrent requests can implement a throttling strategy that temporarily rejects any further requests until its workload has eased. An application trying to access the database might fail to connect, but if it tries again after a delay it might succeed.

Solution

In the cloud, transient faults aren't uncommon and an application should be designed to handle them elegantly and transparently. This minimizes the effects faults can have on the business tasks the application is performing.

If an application detects a failure when it tries to send a request to a remote service, it can handle the failure using the following strategies:

- **Cancel**. If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted.
- **Retry**. If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated and the request will probably be successful.
- **Retry after delay**. If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable time before retrying the request.

For the more common transient failures, the period between retries should be chosen to spread requests from multiple instances of the application as evenly as possible. This reduces the chance of a busy service continuing to be overloaded. If many instances of an application are continually overwhelming a service with retry requests, it'll take the service longer to recover.

If the request still fails, the application can wait and make another attempt. If necessary, this process can be repeated with increasing delays between retry attempts, until some maximum number of requests have been attempted. The delay can be increased incrementally or exponentially, depending on the type of failure and the probability that it'll be corrected during this time.

The following diagram illustrates invoking an operation in a hosted service using this pattern. If the request is unsuccessful after a predefined number of attempts, the application should treat the fault as an exception and handle it accordingly.

1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

The application should wrap all attempts to access a remote service in code that implements a retry policy matching one of the strategies listed above. Requests sent to different services can be subject to different policies. Some vendors provide libraries that implement retry policies, where the application can specify the maximum number of retries, the time between retry attempts, and other parameters.

An application should log the details of faults and failing operations. This information is useful to operators. That being said, in order to avoid flooding operators with alerts on operations where subsequently retried attempts were successful, it is best to log early failures as *informational entries* and only the failure of the last of the retry attempts as an actual error. Here is an example of how this logging model would look like.

If a service is frequently unavailable or busy, it's often because the service has exhausted its resources. You can reduce the frequency of these faults by scaling out the service. For example, if a database service is continually overloaded, it might be beneficial to partition the database and spread the load across multiple servers.

> Microsoft Entity Framework provides facilities for retrying database operations. Also, most Azure services and client SDKs include a retry mechanism. For more information, see Retry guidance for specific services.

Issues and considerations

You should consider the following points when deciding how to implement this pattern.

The retry policy should be tuned to match the business requirements of the application and the nature of the failure. For some noncritical operations, it's better to fail fast rather than retry several times and impact the throughput of the application. For example, in an interactive web application accessing a remote service, it's better to fail after a smaller number of retries with only a short delay between retry attempts, and display a suitable message to the user (for example, "please try again later"). For a batch application, it might be more appropriate to increase the number of retry attempts with an exponentially increasing delay between attempts.

An aggressive retry policy with minimal delay between attempts, and a large number of retries, could further degrade a busy service that's running close to or at capacity. This retry policy could also affect the responsiveness of the application if it's continually trying to perform a failing operation.

If a request still fails after a significant number of retries, it's better for the application to prevent further requests going to the same resource and simply report a failure immediately. When the period expires, the application can tentatively allow one or more requests through to see whether they're successful. For more details of this strategy, see the Circuit Breaker pattern.

Consider whether the operation is idempotent. If so, it's inherently safe to retry. Otherwise, retries could cause the operation to be executed more than once, with unintended side effects. For example, a service might receive

the request, process the request successfully, but fail to send a response. At that point, the retry logic might re-send the request, assuming that the first request wasn't received.

A request to a service can fail for a variety of reasons raising different exceptions depending on the nature of the failure. Some exceptions indicate a failure that can be resolved quickly, while others indicate that the failure is longer lasting. It's useful for the retry policy to adjust the time between retry attempts based on the type of the exception.

Consider how retrying an operation that's part of a transaction will affect the overall transaction consistency. Fine tune the retry policy for transactional operations to maximize the chance of success and reduce the need to undo all the transaction steps.

Ensure that all retry code is fully tested against a variety of failure conditions. Check that it doesn't severely impact the performance or reliability of the application, cause excessive load on services and resources, or generate race conditions or bottlenecks.

Implement retry logic only where the full context of a failing operation is understood. For example, if a task that contains a retry policy invokes another task that also contains a retry policy, this extra layer of retries can add long delays to the processing. It might be better to configure the lower-level task to fail fast and report the reason for the failure back to the task that invoked it. This higher-level task can then handle the failure based on its own policy.

It's important to log all connectivity failures that cause a retry so that underlying problems with the application, services, or resources can be identified.

Investigate the faults that are most likely to occur for a service or a resource to discover if they're likely to be long lasting or terminal. If they are, it's better to handle the fault as an exception. The application can report or log the exception, and then try to continue either by invoking an alternative service (if one is available), or by offering degraded functionality. For more information on how to detect and handle long-lasting faults, see the Circuit Breaker pattern.

When to use this pattern

Use this pattern when an application could experience transient faults as it interacts with a remote service or accesses a remote resource. These faults are expected to be short lived, and repeating a request that has previously failed could succeed on a subsequent attempt.

This pattern might not be useful:

- When a fault is likely to be long lasting, because this can affect the responsiveness of an application. The application might be wasting time and resources trying to repeat a request that's likely to fail.
- For handling failures that aren't due to transient faults, such as internal exceptions caused by errors in the business logic of an application.
- As an alternative to addressing scalability issues in a system. If an application experiences frequent busy faults, it's often a sign that the service or resource being accessed should be scaled up.

Example

This example in C# illustrates an implementation of the Retry pattern. The `OperationWithBasicRetryAsync` method, shown below, invokes an external service asynchronously through the `TransientOperationAsync` method. The details of the `TransientOperationAsync` method will be specific to the service and are omitted from the sample code.

```
private int retryCount = 3;
private readonly TimeSpan delay = TimeSpan.FromSeconds(5);

public async Task OperationWithBasicRetryAsync()
{
  int currentRetry = 0;

  for (;;)
  {
    try
    {
      // Call external service.
      await TransientOperationAsync();

      // Return or break.
      break;
    }
    catch (Exception ex)
    {
      Trace.TraceError("Operation Exception");

      currentRetry++;

      // Check if the exception thrown was a transient exception
      // based on the logic in the error detection strategy.
      // Determine whether to retry the operation, as well as how
      // long to wait, based on the retry strategy.
      if (currentRetry > this.retryCount || !IsTransient(ex))
      {
        // If this isn't a transient error or we shouldn't retry,
        // rethrow the exception.
        throw;
      }
    }

    // Wait to retry the operation.
    // Consider calculating an exponential delay here and
    // using a strategy best suited for the operation and fault.
    await Task.Delay(delay);
  }
}

// Async method that wraps a call to a remote service (details not shown).
private async Task TransientOperationAsync()
{
  ...
}
```

The statement that invokes this method is contained in a try/catch block wrapped in a for loop. The for loop exits if the call to the `TransientOperationAsync` method succeeds without throwing an exception. If the `TransientOperationAsync` method fails, the catch block examines the reason for the failure. If it's believed to be a transient error the code waits for a short delay before retrying the operation.

The for loop also tracks the number of times that the operation has been attempted, and if the code fails three times the exception is assumed to be more long lasting. If the exception isn't transient or it's long lasting, the catch handler throws an exception. This exception exits the for loop and should be caught by the code that invokes the `OperationWithBasicRetryAsync` method.

The `IsTransient` method, shown below, checks for a specific set of exceptions that are relevant to the environment the code is run in. The definition of a transient exception will vary according to the resources being accessed and the environment the operation is being performed in.

```
private bool IsTransient(Exception ex)
{
  // Determine if the exception is transient.
  // In some cases this is as simple as checking the exception type, in other
  // cases it might be necessary to inspect other properties of the exception.
  if (ex is OperationTransientException)
    return true;

  var webException = ex as WebException;
  if (webException != null)
  {
    // If the web exception contains one of the following status values
    // it might be transient.
    return new[] {WebExceptionStatus.ConnectionClosed,
                  WebExceptionStatus.Timeout,
                  WebExceptionStatus.RequestCanceled }.
            Contains(webException.Status);
  }

  // Additional exception checking logic goes here.
  return false;
}
```

Next steps

- For most Azure services, the client SDKs include built-in retry logic. For more information, see Retry guidance for Azure services.
- Before writing custom retry logic, consider using a general framework such as Polly for .NET or Resilience4j for Java.

Related guidance

- Circuit Breaker pattern. If a failure is expected to be more long lasting, it might be more appropriate to implement the Circuit Breaker pattern. Combining the Retry and Circuit Breaker patterns provides a comprehensive approach to handling faults.
- When processing commands that change business data, be aware that retries can result in the action being performed twice, which could be problematic if that action is something like charging a customer's credit card. Using the Idempotence pattern described in this blog post can help deal with these situations.

Scheduler Agent Supervisor pattern

11/19/2021 • 16 minutes to read • Edit Online

Coordinate a set of distributed actions as a single operation. If any of the actions fail, try to handle the failures transparently, or else undo the work that was performed, so the entire operation succeeds or fails as a whole. This can add resiliency to a distributed system, by enabling it to recover and retry actions that fail due to transient exceptions, long-lasting faults, and process failures.

Context and problem

An application performs tasks that include a number of steps, some of which might invoke remote services or access remote resources. The individual steps might be independent of each other, but they are orchestrated by the application logic that implements the task.

Whenever possible, the application should ensure that the task runs to completion and resolve any failures that might occur when accessing remote services or resources. Failures can occur for many reasons. For example, the network might be down, communications could be interrupted, a remote service might be unresponsive or in an unstable state, or a remote resource might be temporarily inaccessible, perhaps due to resource constraints. In many cases the failures will be transient and can be handled by using the Retry pattern.

If the application detects a more permanent fault it can't easily recover from, it must be able to restore the system to a consistent state and ensure integrity of the entire operation.

Solution

The Scheduler Agent Supervisor pattern defines the following actors. These actors orchestrate the steps to be performed as part of the overall task.

- The **Scheduler** arranges for the steps that make up the task to be executed and orchestrates their operation. These steps can be combined into a pipeline or workflow. The Scheduler is responsible for ensuring that the steps in this workflow are performed in the right order. As each step is performed, the Scheduler records the state of the workflow, such as "step not yet started," "step running," or "step completed." The state information should also include an upper limit of the time allowed for the step to finish, called the complete-by time. If a step requires access to a remote service or resource, the Scheduler invokes the appropriate Agent, passing it the details of the work to be performed. The Scheduler typically communicates with an Agent using asynchronous request/response messaging. This can be implemented using queues, although other distributed messaging technologies could be used instead.

  > The Scheduler performs a similar function to the Process Manager in the Process Manager pattern. The actual workflow is typically defined and implemented by a workflow engine that's controlled by the Scheduler. This approach decouples the business logic in the workflow from the Scheduler.

- The **Agent** contains logic that encapsulates a call to a remote service, or access to a remote resource referenced by a step in a task. Each Agent typically wraps calls to a single service or resource, implementing the appropriate error handling and retry logic (subject to a timeout constraint, described later). If the steps in the workflow being run by the Scheduler use several services and resources across different steps, each step might reference a different Agent (this is an implementation detail of the pattern).
- The **Supervisor** monitors the status of the steps in the task being performed by the Scheduler. It runs

periodically (the frequency will be system-specific), and examines the status of steps maintained by the Scheduler. If it detects any that have timed out or failed, it arranges for the appropriate Agent to recover the step or execute the appropriate remedial action (this might involve modifying the status of a step). Note that the recovery or remedial actions are implemented by the Scheduler and Agents. The Supervisor should simply request that these actions be performed.

The Scheduler, Agent, and Supervisor are logical components and their physical implementation depends on the technology being used. For example, several logical agents might be implemented as part of a single web service.

The Scheduler maintains information about the progress of the task and the state of each step in a durable data store, called the state store. The Supervisor can use this information to help determine whether a step has failed. The figure illustrates the relationship between the Scheduler, the Agents, the Supervisor, and the state store.

> **NOTE**
>
> This diagram shows a simplified version of the pattern. In a real implementation, there might be many instances of the Scheduler running concurrently, each a subset of tasks. Similarly, the system could run multiple instances of each Agent, or even multiple Supervisors. In this case, Supervisors must coordinate their work with each other carefully to ensure that they don't compete to recover the same failed steps and tasks. The Leader Election pattern provides one possible solution to this problem.

When the application is ready to run a task, it submits a request to the Scheduler. The Scheduler records initial state information about the task and its steps (for example, step not yet started) in the state store and then starts performing the operations defined by the workflow. As the Scheduler starts each step, it updates the information about the state of that step in the state store (for example, step running).

If a step references a remote service or resource, the Scheduler sends a message to the appropriate Agent. The message contains the information that the Agent needs to pass to the service or access the resource, in addition

to the complete-by time for the operation. If the Agent completes its operation successfully, it returns a response to the Scheduler. The Scheduler can then update the state information in the state store (for example, step completed) and perform the next step. This process continues until the entire task is complete.

An Agent can implement any retry logic that's necessary to perform its work. However, if the Agent doesn't complete its work before the complete-by period expires, the Scheduler will assume that the operation has failed. In this case, the Agent should stop its work and not try to return anything to the Scheduler (not even an error message), or try any form of recovery. The reason for this restriction is that, after a step has timed out or failed, another instance of the Agent might be scheduled to run the failing step (this process is described later).

If the Agent fails, the Scheduler won't receive a response. The pattern doesn't make a distinction between a step that has timed out and one that has genuinely failed.

If a step times out or fails, the state store will contain a record that indicates that the step is running, but the complete-by time will have passed. The Supervisor looks for steps like this and tries to recover them. One possible strategy is for the Supervisor to update the complete-by value to extend the time available to complete the step, and then send a message to the Scheduler identifying the step that has timed out. The Scheduler can then try to repeat this step. However, this design requires the tasks to be idempotent.

The Supervisor might need to prevent the same step from being retried if it continually fails or times out. To do this, the Supervisor could maintain a retry count for each step, along with the state information, in the state store. If this count exceeds a predefined threshold the Supervisor can adopt a strategy of waiting for an extended period before notifying the Scheduler that it should retry the step, in the expectation that the fault will be resolved during this period. Alternatively, the Supervisor can send a message to the Scheduler to request the entire task be undone by implementing a Compensating Transaction pattern. This approach will depend on the Scheduler and Agents providing the information necessary to implement the compensating operations for each step that completed successfully.

> It isn't the purpose of the Supervisor to monitor the Scheduler and Agents, and restart them if they fail. This aspect of the system should be handled by the infrastructure these components are running in. Similarly, the Supervisor shouldn't have knowledge of the actual business operations that the tasks being performed by the Scheduler are running (including how to compensate should these tasks fail). This is the purpose of the workflow logic implemented by the Scheduler. The sole responsibility of the Supervisor is to determine whether a step has failed and arrange either for it to be repeated or for the entire task containing the failed step to be undone.

If the Scheduler is restarted after a failure, or the workflow being performed by the Scheduler terminates unexpectedly, the Scheduler should be able to determine the status of any inflight task that it was handling when it failed, and be prepared to resume this task from that point. The implementation details of this process are likely to be system-specific. If the task can't be recovered, it might be necessary to undo the work already performed by the task. This might also require implementing a compensating transaction.

The key advantage of this pattern is that the system is resilient in the event of unexpected temporary or unrecoverable failures. The system can be constructed to be self-healing. For example, if an Agent or the Scheduler fails, a new one can be started and the Supervisor can arrange for a task to be resumed. If the Supervisor fails, another instance can be started and can take over from where the failure occurred. If the Supervisor is scheduled to run periodically, a new instance can be automatically started after a predefined interval. The state store can be replicated to reach an even greater degree of resiliency.

Issues and considerations

You should consider the following points when deciding how to implement this pattern:

- This pattern can be difficult to implement and requires thorough testing of each possible failure mode of the system.

- The recovery/retry logic implemented by the Scheduler is complex and dependent on state information held in the state store. It might also be necessary to record the information required to implement a compensating transaction in a durable data store.
- How often the Supervisor runs will be important. It should run often enough to prevent any failed steps from blocking an application for an extended period, but it shouldn't run so often that it becomes an overhead.
- The steps performed by an Agent could be run more than once. The logic that implements these steps should be idempotent.

When to use this pattern

Use this pattern when a process that runs in a distributed environment, such as the cloud, must be resilient to communications failure and/or operational failure.

This pattern might not be suitable for tasks that don't invoke remote services or access remote resources.

Example

A web application that implements an ecommerce system has been deployed on Microsoft Azure. Users can run this application to browse the available products and to place orders. The user interface runs as a web role, and the order processing elements of the application are implemented as a set of worker roles. Part of the order processing logic involves accessing a remote service, and this aspect of the system could be prone to transient or more long-lasting faults. For this reason, the designers used the Scheduler Agent Supervisor pattern to implement the order processing elements of the system.

When a customer places an order, the application constructs a message that describes the order and posts this message to a queue. A separate submission process, running in a worker role, retrieves the message, inserts the order details into the orders database, and creates a record for the order process in the state store. Note that the inserts into the orders database and the state store are performed as part of the same operation. The submission process is designed to ensure that both inserts complete together.

The state information that the submission process creates for the order includes:

- **OrderID**. The ID of the order in the orders database.
- **LockedBy**. The instance ID of the worker role handling the order. There might be multiple current instances of the worker role running the Scheduler, but each order should only be handled by a single instance.
- **CompleteBy**. The time the order should be processed by.
- **ProcessState**. The current state of the task handling the order. The possible states are:
  - **Pending**. The order has been created but processing hasn't yet been started.
  - **Processing**. The order is currently being processed.
  - **Processed**. The order has been processed successfully.
  - **Error**. The order processing has failed.
- **FailureCount**. The number of times that processing has been tried for the order.

In this state information, the `OrderID` field is copied from the order ID of the new order. The `LockedBy` and `CompleteBy` fields are set to `null`, the `ProcessState` field is set to `Pending`, and the `FailureCount` field is set to 0.

> **NOTE**
> In this example, the order handling logic is relatively simple and only has a single step that invokes a remote service. In a more complex multistep scenario, the submission process would likely involve several steps, and so several records would be created in the state store — each one describing the state of an individual step.

The Scheduler also runs as part of a worker role and implements the business logic that handles the order. An instance of the Scheduler polling for new orders examines the state store for records where the `LockedBy` field is null and the `ProcessState` field is pending. When the Scheduler finds a new order, it immediately populates the `LockedBy` field with its own instance ID, sets the `CompleteBy` field to an appropriate time, and sets the `ProcessState` field to processing. The code is designed to be exclusive and atomic to ensure that two concurrent instances of the Scheduler can't try to handle the same order simultaneously.

The Scheduler then runs the business workflow to process the order asynchronously, passing it the value in the `OrderID` field from the state store. The workflow handling the order retrieves the details of the order from the orders database and performs its work. When a step in the order processing workflow needs to invoke the remote service, it uses an Agent. The workflow step communicates with the Agent using a pair of Azure Service Bus message queues acting as a request/response channel. The figure shows a high-level view of the solution.

The message sent to the Agent from a workflow step describes the order and includes the complete-by time. If the Agent receives a response from the remote service before the complete-by time expires, it posts a reply message on the Service Bus queue on which the workflow is listening. When the workflow step receives the valid reply message, it completes its processing and the Scheduler sets the `ProcessState` field of the order state to processed. At this point, the order processing has completed successfully.

If the complete-by time expires before the Agent receives a response from the remote service, the Agent simply halts its processing and terminates handling the order. Similarly, if the workflow handling the order exceeds the complete-by time, it also terminates. In both cases, the state of the order in the state store remains set to processing, but the complete-by time indicates that the time for processing the order has passed and the process is deemed to have failed. Note that if the Agent that's accessing the remote service, or the workflow that's handling the order (or both) terminate unexpectedly, the information in the state store will again remain set to processing and eventually will have an expired complete-by value.

If the Agent detects an unrecoverable, nontransient fault while it's trying to contact the remote service, it can send an error response back to the workflow. The Scheduler can set the status of the order to error and raise an event that alerts an operator. The operator can then try to resolve the reason for the failure manually and resubmit the failed processing step.

The Supervisor periodically examines the state store looking for orders with an expired complete-by value. If the Supervisor finds a record, it increments the `FailureCount` field. If the failure count value is below a specified threshold value, the Supervisor resets the `LockedBy` field to null, updates the `CompleteBy` field with a new expiration time, and sets the `ProcessState` field to pending. An instance of the Scheduler can pick up this order and perform its processing as before. If the failure count value exceeds a specified threshold, the reason for the failure is assumed to be nontransient. The Supervisor sets the status of the order to error and raises an event that alerts an operator.

> In this example, the Supervisor is implemented in a separate worker role. You can use a variety of strategies to arrange for the Supervisor task to be run, including using the Azure Scheduler service (not to be confused with the Scheduler component in this pattern). For more information about the Azure Scheduler service, visit the Scheduler page.

Although it isn't shown in this example, the Scheduler might need to keep the application that submitted the order informed about the progress and status of the order. The application and the Scheduler are isolated from each other to eliminate any dependencies between them. The application has no knowledge of which instance of the Scheduler is handling the order, and the Scheduler is unaware of which specific application instance posted the order.

To allow the order status to be reported, the application could use its own private response queue. The details of this response queue would be included as part of the request sent to the submission process, which would include this information in the state store. The Scheduler would then post messages to this queue indicating the status of the order (request received, order completed, order failed, and so on). It should include the order ID in these messages so they can be correlated with the original request by the application.

Next steps

The following guidance might also be relevant when implementing this pattern:

- Asynchronous Messaging Primer. The components in the Scheduler Agent Supervisor pattern typically run decoupled from each other and communicate asynchronously. Describes some of the approaches that can be used to implement asynchronous communication based on message queues.
- Reference 6: A Saga on Sagas. An example showing how the CQRS pattern uses a process manager (part of the CQRS Journey guidance).

- Microsoft Azure Scheduler

Related guidance

The following patterns might also be relevant when implementing this pattern:

- Retry pattern. An Agent can use this pattern to transparently retry an operation that accesses a remote service or resource that has previously failed. Use when the expectation is that the cause of the failure is transient and can be corrected.
- Circuit Breaker pattern. An Agent can use this pattern to handle faults that take a variable amount of time to correct when connecting to a remote service or resource.
- Compensating Transaction pattern. If the workflow being performed by a Scheduler can't be completed successfully, it might be necessary to undo any work it's previously performed. The Compensating Transaction pattern describes how this can be achieved for operations that follow the eventual consistency model. These types of operations are commonly implemented by a Scheduler that performs complex business processes and workflows.
- Leader Election pattern. It might be necessary to coordinate the actions of multiple instances of a Supervisor to prevent them from attempting to recover the same failed process. The Leader Election pattern describes how to do this.
- Cloud Architecture: The Scheduler-Agent-Supervisor pattern on Clemens Vasters' blog
- Process Manager pattern

Sequential Convoy pattern

11/19/2021 • 3 minutes to read • Edit Online

Process a set of related messages in a defined order, without blocking processing of other groups of messages.

Context and problem

Applications often need to process a sequence of messages in the order they arrive, while still being able to scale out to handle increased load. In a distributed architecture, processing these messages in order is not straightforward, because the workers can scale independently and often pull messages independently, using a Competing Consumers pattern.

For example, an order tracking system receives a ledger containing orders and the relevant operations on those orders. These operations could be to create an order, add a transaction to the order, modify a past transaction, or delete an order. In this system, operations must be performed in a first-in-first-out (FIFO) manner, but only at the order level. However, the initial queue receives a ledger containing transactions for many orders, which may be interleaved.

Solution

Push related messages into categories within the queuing system, and have the queue listeners lock and pull only from one category, one message at a time.

Here's what the general Sequential Convoy pattern looks like:

In the queue, messages for different categories may be interleaved, as shown in the following diagram:

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Category/scale unit. What property of your incoming messages can you scale out on? In the order tracking scenario, this property is the order ID.
- Throughput. What is your target message throughput? If it is very high, you may need to reconsider your FIFO requirements. For example, can you enforce a start/end message, sort by time, then send a batch for processing?
- Service capabilities. Does your choice of message bus allow for one-at-a-time processing of messages within a queue or category of a queue?
- Evolvability. How will you add a new category of message to the system? For example, suppose the ledger system described above is specific one customer. If you needed to onboard a new customer, could you have a set of ledger processors that distribute work per customer ID?
- It's possible that consumers might receive a message out of order, due to variable network latency when sending messages. Consider using sequence numbers to verify ordering. You might also include a special "end of sequence" flag in the last message of a transaction. Stream processing technologies such as Spark or Azure Stream Analytics can process messages in order within a time window.

When to use this pattern

Use this pattern when:

- You have messages that arrive in order and must be processed in the same order.
- Arriving messages are or can be "categorized" in such a way that the category becomes a unit of scale for the system.

This pattern might not be suitable for:

- Extremely high throughput scenarios (millions of messages/minute or second), as the FIFO requirement limits the scaling that can be done by the system.

Example

On Azure, this pattern can be implemented using Azure Service Bus message sessions. For the consumers, you can use either Logic Apps with the Service Bus peek-lock connector or Azure Functions with the Service Bus trigger.

For the previous order-tracking example, process each ledger message in the order it's received, and send each transaction to another queue where the category is set to the order ID. A transaction will never span multiple orders in this scenario, so consumers process each category in parallel but FIFO within the category.

The ledge processor fan outs the messages by de-batching the content of each message in the first queue:

The ledger processor takes care of:

1. Walking the ledger one transaction at a time.
2. Setting the session ID of the message to match the order ID.
3. Sending each ledger transaction to a secondary queue with the session ID set to the order ID.

The consumers listen to the secondary queue where they process all messages with matching order IDs *in order*

from the queue. Consumers use peek-lock mode.

When considering scalability, the ledger queue is a primary bottleneck. Different transactions posted to the ledger could reference the same order ID. However, messages can fan out *after* the ledger to the number of orders in a serverless environment.

Next steps

The following information may be relevant when implementing this pattern:

- Message sessions: first in, first out (FIFO)
- Peek-Lock Message (Non-Destructive Read)
- In order delivery of correlated messages in Logic Apps by using Service Bus sessions (MSDN blog)

Sharding pattern

11/19/2021 • 19 minutes to read • Edit Online

Divide a data store into a set of horizontal partitions or shards. This can improve scalability when storing and accessing large volumes of data.

Context and problem

A data store hosted by a single server might be subject to the following limitations:

- **Storage space**. A data store for a large-scale cloud application is expected to contain a huge volume of data that could increase significantly over time. A server typically provides only a finite amount of disk storage, but you can replace existing disks with larger ones, or add further disks to a machine as data volumes grow. However, the system will eventually reach a limit where it isn't possible to easily increase the storage capacity on a given server.
- **Computing resources**. A cloud application is required to support a large number of concurrent users, each of which run queries that retrieve information from the data store. A single server hosting the data store might not be able to provide the necessary computing power to support this load, resulting in extended response times for users and frequent failures as applications attempting to store and retrieve data time out. It might be possible to add memory or upgrade processors, but the system will reach a limit when it isn't possible to increase the compute resources any further.
- **Network bandwidth**. Ultimately, the performance of a data store running on a single server is governed by the rate the server can receive requests and send replies. It's possible that the volume of network traffic might exceed the capacity of the network used to connect to the server, resulting in failed requests.
- **Geography**. It might be necessary to store data generated by specific users in the same region as those users for legal, compliance, or performance reasons, or to reduce latency of data access. If the users are dispersed across different countries or regions, it might not be possible to store the entire data for the application in a single data store.

Scaling vertically by adding more disk capacity, processing power, memory, and network connections can postpone the effects of some of these limitations, but it's likely to only be a temporary solution. A commercial cloud application capable of supporting large numbers of users and high volumes of data must be able to scale almost indefinitely, so vertical scaling isn't necessarily the best solution.

Solution

Divide the data store into horizontal partitions or shards. Each shard has the same schema, but holds its own distinct subset of the data. A shard is a data store in its own right (it can contain the data for many entities of different types), running on a server acting as a storage node.

This pattern has the following benefits:

- You can scale the system out by adding further shards running on additional storage nodes.
- A system can use off-the-shelf hardware rather than specialized and expensive computers for each storage node.
- You can reduce contention and improve performance by balancing the workload across shards.
- In the cloud, shards can be located physically close to the users that'll access the data.

When dividing a data store up into shards, decide which data should be placed in each shard. A shard typically contains items that fall within a specified range determined by one or more attributes of the data. These attributes form the shard key (sometimes referred to as the partition key). The shard key should be static. It shouldn't be based on data that might change.

Sharding physically organizes the data. When an application stores and retrieves data, the sharding logic directs the application to the appropriate shard. This sharding logic can be implemented as part of the data access code in the application, or it could be implemented by the data storage system if it transparently supports sharding.

Abstracting the physical location of the data in the sharding logic provides a high level of control over which shards contain which data. It also enables data to migrate between shards without reworking the business logic of an application if the data in the shards need to be redistributed later (for example, if the shards become unbalanced). The tradeoff is the additional data access overhead required in determining the location of each data item as it's retrieved.

To ensure optimal performance and scalability, it's important to split the data in a way that's appropriate for the types of queries that the application performs. In many cases, it's unlikely that the sharding scheme will exactly match the requirements of every query. For example, in a multi-tenant system an application might need to retrieve tenant data using the tenant ID, but it might also need to look up this data based on some other attribute such as the tenant's name or location. To handle these situations, implement a sharding strategy with a shard key that supports the most commonly performed queries.

If queries regularly retrieve data using a combination of attribute values, you can likely define a composite shard key by linking attributes together. Alternatively, use a pattern such as Index Table to provide fast lookup to data based on attributes that aren't covered by the shard key.

Sharding strategies

Three strategies are commonly used when selecting the shard key and deciding how to distribute data across shards. Note that there doesn't have to be a one-to-one correspondence between shards and the servers that host them—a single server can host multiple shards. The strategies are:

The Lookup strategy. In this strategy the sharding logic implements a map that routes a request for data to the shard that contains that data using the shard key. In a multi-tenant application all the data for a tenant might be stored together in a shard using the tenant ID as the shard key. Multiple tenants might share the same shard, but the data for a single tenant won't be spread across multiple shards. The figure illustrates sharding tenant data based on tenant IDs.

The mapping between the shard key and the physical storage can be based on physical shards where each shard key maps to a physical partition. Alternatively, a more flexible technique for rebalancing shards is virtual partitioning, where shard keys map to the same number of virtual shards, which in turn map to fewer physical partitions. In this approach, an application locates data using a shard key that refers to a virtual shard, and the system transparently maps virtual shards to physical partitions. The mapping between a virtual shard and a physical partition can change without requiring the application code be modified to use a different set of shard keys.

The Range strategy. This strategy groups related items together in the same shard, and orders them by shard key—the shard keys are sequential. It's useful for applications that frequently retrieve sets of items using range queries (queries that return a set of data items for a shard key that falls within a given range). For example, if an application regularly needs to find all orders placed in a given month, this data can be retrieved more quickly if all orders for a month are stored in date and time order in the same shard. If each order was stored in a different shard, they'd have to be fetched individually by performing a large number of point queries (queries that return a single data item). The next figure illustrates storing sequential sets (ranges) of data in shard.

In this example, the shard key is a composite key containing the order month as the most significant element, followed by the order day and the time. The data for orders is naturally sorted when new orders are created and added to a shard. Some data stores support two-part shard keys containing a partition key element that identifies the shard and a row key that uniquely identifies an item in the shard. Data is usually held in row key order in the shard. Items that are subject to range queries and need to be grouped together can use a shard key that has the same value for the partition key but a unique value for the row key.

The Hash strategy. The purpose of this strategy is to reduce the chance of hotspots (shards that receive a disproportionate amount of load). It distributes the data across the shards in a way that achieves a balance between the size of each shard and the average load that each shard will encounter. The sharding logic computes the shard to store an item in based on a hash of one or more attributes of the data. The chosen hashing function should distribute data evenly across the shards, possibly by introducing some random element into the computation. The next figure illustrates sharding tenant data based on a hash of tenant IDs.

To understand the advantage of the Hash strategy over other sharding strategies, consider how a multi-tenant application that enrolls new tenants sequentially might assign the tenants to shards in the data store. When using the Range strategy, the data for tenants 1 to n will all be stored in shard A, the data for tenants n+1 to m will all be stored in shard B, and so on. If the most recently registered tenants are also the most active, most data activity will occur in a small number of shards, which could cause hotspots. In contrast, the Hash strategy allocates tenants to shards based on a hash of their tenant ID. This means that sequential tenants are most likely to be allocated to different shards, which will distribute the load across them. The previous figure shows this for tenants 55 and 56.

The three sharding strategies have the following advantages and considerations:

- **Lookup**. This offers more control over the way that shards are configured and used. Using virtual shards reduces the impact when rebalancing data because new physical partitions can be added to even out the workload. The mapping between a virtual shard and the physical partitions that implement the shard can be modified without affecting application code that uses a shard key to store and retrieve data. Looking up shard locations can impose an additional overhead.
- **Range**. This is easy to implement and works well with range queries because they can often fetch multiple data items from a single shard in a single operation. This strategy offers easier data management. For example, if users in the same region are in the same shard, updates can be scheduled in each time zone based on the local load and demand pattern. However, this strategy doesn't provide optimal balancing between shards. Rebalancing shards is difficult and might not resolve the problem of uneven load if the majority of activity is for adjacent shard keys.
- **Hash**. This strategy offers a better chance of more even data and load distribution. Request routing can be accomplished directly by using the hash function. There's no need to maintain a map. Note that computing the hash might impose an additional overhead. Also, rebalancing shards is difficult.

Most common sharding systems implement one of the approaches described above, but you should also consider the business requirements of your applications and their patterns of data usage. For example, in a multi-tenant application:

- You can shard data based on workload. You could segregate the data for highly volatile tenants in separate shards. The speed of data access for other tenants might be improved as a result.
- You can shard data based on the location of tenants. You can take the data for tenants in a specific geographic region offline for backup and maintenance during off-peak hours in that region, while the data for tenants in other regions remains online and accessible during their business hours.
- High-value tenants could be assigned their own private, high performing, lightly loaded shards, whereas lower-value tenants might be expected to share more densely-packed, busy shards.
- The data for tenants that need a high degree of data isolation and privacy can be stored on a completely separate server.

Scaling and data movement operations

Each of the sharding strategies implies different capabilities and levels of complexity for managing scale in, scale out, data movement, and maintaining state.

The Lookup strategy permits scaling and data movement operations to be carried out at the user level, either online or offline. The technique is to suspend some or all user activity (perhaps during off-peak periods), move the data to the new virtual partition or physical shard, change the mappings, invalidate or refresh any caches that hold this data, and then allow user activity to resume. Often this type of operation can be centrally managed. The Lookup strategy requires state to be highly cacheable and replica friendly.

The Range strategy imposes some limitations on scaling and data movement operations, which must typically

be carried out when a part or all of the data store is offline because the data must be split and merged across the shards. Moving the data to rebalance shards might not resolve the problem of uneven load if the majority of activity is for adjacent shard keys or data identifiers that are within the same range. The Range strategy might also require some state to be maintained in order to map ranges to the physical partitions.

The Hash strategy makes scaling and data movement operations more complex because the partition keys are hashes of the shard keys or data identifiers. The new location of each shard must be determined from the hash function, or the function modified to provide the correct mappings. However, the Hash strategy doesn't require maintenance of state.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- Sharding is complementary to other forms of partitioning, such as vertical partitioning and functional partitioning. For example, a single shard can contain entities that have been partitioned vertically, and a functional partition can be implemented as multiple shards. For more information about partitioning, see the Data Partitioning Guidance.
- Keep shards balanced so they all handle a similar volume of I/O. As data is inserted and deleted, it's necessary to periodically rebalance the shards to guarantee an even distribution and to reduce the chance of hotspots. Rebalancing can be an expensive operation. To reduce the necessity of rebalancing, plan for growth by ensuring that each shard contains sufficient free space to handle the expected volume of changes. You should also develop strategies and scripts you can use to quickly rebalance shards if this becomes necessary.
- Use stable data for the shard key. If the shard key changes, the corresponding data item might have to move between shards, increasing the amount of work performed by update operations. For this reason, avoid basing the shard key on potentially volatile information. Instead, look for attributes that are invariant or that naturally form a key.
- Ensure that shard keys are unique. For example, avoid using autoincrementing fields as the shard key. In some systems, autoincremented fields can't be coordinated across shards, possibly resulting in items in different shards having the same shard key.

  > Autoincremented values in other fields that are not shard keys can also cause problems. For example, if you use autoincremented fields to generate unique IDs, then two different items located in different shards might be assigned the same ID.

- It might not be possible to design a shard key that matches the requirements of every possible query against the data. Shard the data to support the most frequently performed queries, and if necessary create secondary index tables to support queries that retrieve data using criteria based on attributes that aren't part of the shard key. For more information, see the Index Table pattern.
- Queries that access only a single shard are more efficient than those that retrieve data from multiple shards, so avoid implementing a sharding system that results in applications performing large numbers of queries that join data held in different shards. Remember that a single shard can contain the data for multiple types of entities. Consider denormalizing your data to keep related entities that are commonly queried together (such as the details of customers and the orders that they have placed) in the same shard to reduce the number of separate reads that an application performs.

  > If an entity in one shard references an entity stored in another shard, include the shard key for the second entity as part of the schema for the first entity. This can help to improve the performance of queries that reference related data across shards.

- If an application must perform queries that retrieve data from multiple shards, it might be possible to fetch this data by using parallel tasks. Examples include fan-out queries, where data from multiple shards is retrieved in parallel and then aggregated into a single result. However, this approach inevitably adds some complexity to the data access logic of a solution.
- For many applications, creating a larger number of small shards can be more efficient than having a small number of large shards because they can offer increased opportunities for load balancing. This can also be useful if you anticipate the need to migrate shards from one physical location to another. Moving a small shard is quicker than moving a large one.
- Make sure the resources available to each shard storage node are sufficient to handle the scalability requirements in terms of data size and throughput. For more information, see the section "Designing Partitions for Scalability" in the Data Partitioning Guidance.
- Consider replicating reference data to all shards. If an operation that retrieves data from a shard also references static or slow-moving data as part of the same query, add this data to the shard. The application can then fetch all of the data for the query easily, without having to make an additional round trip to a separate data store.

  > If reference data held in multiple shards changes, the system must synchronize these changes across all shards. The system can experience a degree of inconsistency while this synchronization occurs. If you do this, you should design your applications to be able to handle it.

- It can be difficult to maintain referential integrity and consistency between shards, so you should minimize operations that affect data in multiple shards. If an application must modify data across shards, evaluate whether complete data consistency is actually required. Instead, a common approach in the cloud is to implement eventual consistency. The data in each partition is updated separately, and the application logic must take responsibility for ensuring that the updates all complete successfully, as well as handling the inconsistencies that can arise from querying data while an eventually consistent operation is running. For more information about implementing eventual consistency, see the Data Consistency Primer.
- Configuring and managing a large number of shards can be a challenge. Tasks such as monitoring, backing up, checking for consistency, and logging or auditing must be accomplished on multiple shards and servers, possibly held in multiple locations. These tasks are likely to be implemented using scripts or other automation solutions, but that might not completely eliminate the additional administrative requirements.
- Shards can be geolocated so that the data that they contain is close to the instances of an application that use it. This approach can considerably improve performance, but requires additional consideration for tasks that must access multiple shards in different locations.

When to use this pattern

Use this pattern when a data store is likely to need to scale beyond the resources available to a single storage node, or to improve performance by reducing contention in a data store.

> **NOTE**
>
> The primary focus of sharding is to improve the performance and scalability of a system, but as a by-product it can also improve availability due to how the data is divided into separate partitions. A failure in one partition doesn't necessarily prevent an application from accessing data held in other partitions, and an operator can perform maintenance or recovery of one or more partitions without making the entire data for an application inaccessible. For more information, see the Data Partitioning Guidance.

Example

The following example in C# uses a set of SQL Server databases acting as shards. Each database holds a subset of the data used by an application. The application retrieves data that's distributed across the shards using its own sharding logic (this is an example of a fan-out query). The details of the data that's located in each shard is returned by a method called `GetShards`. This method returns an enumerable list of `ShardInformation` objects, where the `ShardInformation` type contains an identifier for each shard and the SQL Server connection string that an application should use to connect to the shard (the connection strings aren't shown in the code example).

```
private IEnumerable<ShardInformation> GetShards()
{
  // This retrieves the connection information from a shard store
  // (commonly a root database).
  return new[]
  {
    new ShardInformation
    {
      Id = 1,
      ConnectionString = ...
    },
    new ShardInformation
    {
      Id = 2,
      ConnectionString = ...
    }
  };
}
```

The code below shows how the application uses the list of `ShardInformation` objects to perform a query that fetches data from each shard in parallel. The details of the query aren't shown, but in this example the data that's retrieved contains a string that could hold information such as the name of a customer if the shards contain the details of customers. The results are aggregated into a `ConcurrentBag` collection for processing by the application.

```
// Retrieve the shards as a ShardInformation[] instance.
var shards = GetShards();

var results = new ConcurrentBag<string>();

// Execute the query against each shard in the shard list.
// This list would typically be retrieved from configuration
// or from a root/primary shard store.
Parallel.ForEach(shards, shard =>
{
  // NOTE: Transient fault handling isn't included,
  // but should be incorporated when used in a real world application.
  using (var con = new SqlConnection(shard.ConnectionString))
  {
    con.Open();
    var cmd = new SqlCommand("SELECT ... FROM ...", con);

    Trace.TraceInformation("Executing command against shard: {0}", shard.Id);

    var reader = cmd.ExecuteReader();
    // Read the results in to a thread-safe data structure.
    while (reader.Read())
    {
      results.Add(reader.GetString(0));
    }
  }
});

Trace.TraceInformation("Fanout query complete - Record Count: {0}",
                       results.Count);
```

Next steps

The following guidance might also be relevant when implementing this pattern:

- Data Consistency Primer. It might be necessary to maintain consistency for data distributed across different shards. Summarizes the issues surrounding maintaining consistency over distributed data, and describes the benefits and tradeoffs of different consistency models.
- Data Partitioning Guidance. Sharding a data store can introduce a range of additional issues. Describes these issues in relation to partitioning data stores in the cloud to improve scalability, reduce contention, and optimize performance.

Related guidance

The following patterns might also be relevant when implementing this pattern:

- Index Table pattern. Sometimes it isn't possible to completely support queries just through the design of the shard key. Enables an application to quickly retrieve data from a large data store by specifying a key other than the shard key.
- Materialized View pattern. To maintain the performance of some query operations, it's useful to create materialized views that aggregate and summarize data, especially if this summary data is based on information that's distributed across shards. Describes how to generate and populate these views.

Sidecar pattern

11/19/2021 • 5 minutes to read • Edit Online

Deploy components of an application into a separate process or container to provide isolation and encapsulation. This pattern can also enable applications to be composed of heterogeneous components and technologies.

This pattern is named *Sidecar* because it resembles a sidecar attached to a motorcycle. In the pattern, the sidecar is attached to a parent application and provides supporting features for the application. The sidecar also shares the same lifecycle as the parent application, being created and retired alongside the parent. The sidecar pattern is sometimes referred to as the sidekick pattern and is a decomposition pattern.

Context and Problem

Applications and services often require related functionality, such as monitoring, logging, configuration, and networking services. These peripheral tasks can be implemented as separate components or services.

If they are tightly integrated into the application, they can run in the same process as the application, making efficient use of shared resources. However, this also means they are not well isolated, and an outage in one of these components can affect other components or the entire application. Also, they usually need to be implemented using the same language as the parent application. As a result, the component and the application have close interdependence on each other.

If the application is decomposed into services, then each service can be built using different languages and technologies. While this gives more flexibility, it means that each component has its own dependencies and requires language-specific libraries to access the underlying platform and any resources shared with the parent application. In addition, deploying these features as separate services can add latency to the application. Managing the code and dependencies for these language-specific interfaces can also add considerable complexity, especially for hosting, deployment, and management.

Solution

Co-locate a cohesive set of tasks with the primary application, but place them inside their own process or container, providing a homogeneous interface for platform services across languages.

A sidecar service is not necessarily part of the application, but is connected to it. It goes wherever the parent

application goes. Sidecars are supporting processes or services that are deployed with the primary application. On a motorcycle, the sidecar is attached to one motorcycle, and each motorcycle can have its own sidecar. In the same way, a sidecar service shares the fate of its parent application. For each instance of the application, an instance of the sidecar is deployed and hosted alongside it.

Advantages of using a sidecar pattern include:

- A sidecar is independent from its primary application in terms of runtime environment and programming language, so you don't need to develop one sidecar per language.
- The sidecar can access the same resources as the primary application. For example, a sidecar can monitor system resources used by both the sidecar and the primary application.
- Because of its proximity to the primary application, there's no significant latency when communicating between them.
- Even for applications that don't provide an extensibility mechanism, you can use a sidecar to extend functionality by attaching it as its own process in the same host or sub-container as the primary application.

The sidecar pattern is often used with containers and referred to as a sidecar container or sidekick container.

Issues and Considerations

- Consider the deployment and packaging format you will use to deploy services, processes, or containers. Containers are particularly well suited to the sidecar pattern.
- When designing a sidecar service, carefully decide on the interprocess communication mechanism. Try to use language- or framework-agnostic technologies unless performance requirements make that impractical.
- Before putting functionality into a sidecar, consider whether it would work better as a separate service or a more traditional daemon.
- Also consider whether the functionality could be implemented as a library or using a traditional extension mechanism. Language-specific libraries may have a deeper level of integration and less network overhead.

When to Use this Pattern

Use this pattern when:

- Your primary application uses a heterogeneous set of languages and frameworks. A component located in a sidecar service can be consumed by applications written in different languages using different frameworks.
- A component is owned by a remote team or a different organization.
- A component or feature must be co-located on the same host as the application
- You need a service that shares the overall lifecycle of your main application, but can be independently updated.
- You need fine-grained control over resource limits for a particular resource or component. For example, you may want to restrict the amount of memory a specific component uses. You can deploy the component as a sidecar and manage memory usage independently of the main application.

This pattern may not be suitable:

- When interprocess communication needs to be optimized. Communication between a parent application and sidecar services includes some overhead, notably latency in the calls. This may not be an acceptable trade-off for chatty interfaces.
- For small applications where the resource cost of deploying a sidecar service for each instance is not worth the advantage of isolation.
- When the service needs to scale differently than or independently from the main applications. If so, it may be

better to deploy the feature as a separate service.

Example

The sidecar pattern is applicable to many scenarios. Some common examples:

- Infrastructure API. The infrastructure development team creates a service that's deployed alongside each application, instead of a language-specific client library to access the infrastructure. The service is loaded as a sidecar and provides a common layer for infrastructure services, including logging, environment data, configuration store, discovery, health checks, and watchdog services. The sidecar also monitors the parent application's host environment and process (or container) and logs the information to a centralized service.
- Manage NGINX/HAProxy. Deploy NGINX with a sidecar service that monitors environment state, then updates the NGINX configuration file and recycles the process when a change in state is needed.
- Ambassador sidecar. Deploy an ambassador service as a sidecar. The application calls through the ambassador, which handles request logging, routing, circuit breaking, and other connectivity related features.
- Offload proxy. Place an NGINX proxy in front of a node.js service instance, to handle serving static file content for the service.

Related guidance

- Ambassador pattern

Static Content Hosting pattern

11/19/2021 • 5 minutes to read • Edit Online

Deploy static content to a cloud-based storage service that can deliver them directly to the client. This can reduce the need for potentially expensive compute instances.

Context and problem

Web applications typically include some elements of static content. This static content might include HTML pages and other resources such as images and documents that are available to the client, either as part of an HTML page (such as inline images, style sheets, and client-side JavaScript files) or as separate downloads (such as PDF documents).

Although web servers are optimized for dynamic rendering and output caching, they still have to handle requests to download static content. This consumes processing cycles that could often be put to better use.

Solution

In most cloud hosting environments, you can put some of an application's resources and static pages in a storage service. The storage service can serve requests for these resources, reducing load on the compute resources that handle other web requests. The cost for cloud-hosted storage is typically much less than for compute instances.

When hosting some parts of an application in a storage service, the main considerations are related to deployment of the application and to securing resources that aren't intended to be available to anonymous users.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

- The hosted storage service must expose an HTTP endpoint that users can access to download the static resources. Some storage services also support HTTPS, so it's possible to host resources in storage services that require SSL.
- For maximum performance and availability, consider using a content delivery network (CDN) to cache the contents of the storage container in multiple datacenters around the world. However, you'll likely have to pay for using the CDN.
- Storage accounts are often geo-replicated by default to provide resiliency against events that might affect a datacenter. This means that the IP address might change, but the URL will remain the same.
- When some content is located in a storage account and other content is in a hosted compute instance, it becomes more challenging to deploy and update the application. You might have to perform separate deployments, and version the application and content to manage it more easily—especially when the static content includes script files or UI components. However, if only static resources have to be updated, they can simply be uploaded to the storage account without needing to redeploy the application package.
- Storage services might not support the use of custom domain names. In this case it's necessary to specify the full URL of the resources in links because they'll be in a different domain from the dynamically-generated content containing the links.
- The storage containers must be configured for public read access, but it's vital to ensure that they aren't

configured for public write access to prevent users being able to upload content.

- Consider using a valet key or token to control access to resources that shouldn't be available anonymously. See the Valet Key pattern for more information.

When to use this pattern

This pattern is useful for:

- Minimizing the hosting cost for websites and applications that contain some static resources.
- Minimizing the hosting cost for websites that consist of only static content and resources. Depending on the capabilities of the hosting provider's storage system, it might be possible to entirely host a fully static website in a storage account.
- Exposing static resources and content for applications running in other hosting environments or on-premises servers.
- Locating content in more than one geographical area using a content delivery network that caches the contents of the storage account in multiple datacenters around the world.
- Monitoring costs and bandwidth usage. Using a separate storage account for some or all of the static content allows the costs to be more easily separated from hosting and runtime costs.

This pattern might not be useful in the following situations:

- The application needs to perform some processing on the static content before delivering it to the client. For example, it might be necessary to add a timestamp to a document.
- The volume of static content is very small. The overhead of retrieving this content from separate storage can outweigh the cost benefit of separating it out from the compute resource.

Example

Azure Storage supports serving static content directly from a storage container. Files are served through anonymous access requests. By default, files have a URL in a subdomain of `core.windows.net`, such as `https://contoso.z4.web.core.windows.net/image.png`. You can configure a custom domain name, and use Azure CDN to access the files over HTTPS. For more information, see Static website hosting in Azure Storage.

Static website hosting makes the files available for anonymous access. If you need to control who can access the files, you can store files in Azure blob storage and then generate shared access signatures to limit access.

The links in the pages delivered to the client must specify the full URL of the resource. If the resource is protected with a valet key, such as a shared access signature, this signature must be included in the URL.

A sample application that demonstrates using external storage for static resources is available on GitHub. This sample uses configuration files to specify the storage account and container that holds the static content.

```
<Setting name="StaticContent.StorageConnectionString"
         value="UseDevelopmentStorage=true" />
<Setting name="StaticContent.Container" value="static-content" />
```

The `Settings` class in the file Settings.cs of the StaticContentHosting.Web project contains methods to extract these values and build a string value containing the cloud storage account container URL.

```
public class Settings
{
  public static string StaticContentStorageConnectionString {
    get
    {
      return RoleEnvironment.GetConfigurationSettingValue(
                              "StaticContent.StorageConnectionString");
    }
  }

  public static string StaticContentContainer
  {
    get
    {
      return RoleEnvironment.GetConfigurationSettingValue("StaticContent.Container");
    }
  }

  public static string StaticContentBaseUrl
  {
    get
    {
        var blobServiceClient = new BlobServiceClient(StaticContentStorageConnectionString);

        return string.Format("{0}/{1}", blobServiceClient.Uri.ToString().TrimEnd('/'),
StaticContentContainer.TrimStart('/'));
    }
  }
}
```

The `StaticContentUrlHtmlHelper` class in the file StaticContentUrlHtmlHelper.cs exposes a method named `StaticContentUrl` that generates a URL containing the path to the cloud storage account if the URL passed to it starts with the ASP.NET root path character (~).

```
public static class StaticContentUrlHtmlHelper
{
  public static string StaticContentUrl(this HtmlHelper helper, string contentPath)
  {
    if (contentPath.StartsWith("~"))
    {
      contentPath = contentPath.Substring(1);
    }

    contentPath = string.Format("{0}/{1}", Settings.StaticContentBaseUrl.TrimEnd('/'),
                                contentPath.TrimStart('/'));

    var url = new UrlHelper(helper.ViewContext.RequestContext);

    return url.Content(contentPath);
  }
}
```

The file Index.cshtml in the Views\Home folder contains an image element that uses the `StaticContentUrl` method to create the URL for its `src` attribute.

```
<img src="@Html.StaticContentUrl("~/media/orderedList1.png")" alt="Test Image" />
```

Next steps

- Static Content Hosting sample. A sample application that demonstrates this pattern.
- Valet Key pattern. If the target resources aren't supposed to be available to anonymous users, use this pattern to restrict direct access.
- Serverless web application on Azure. A reference architecture that uses static website hosting with Azure Functions to implement a serverless web app.

Strangler Fig pattern

11/19/2021 • 2 minutes to read • Edit Online

Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services. As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it.

Context and problem

As systems age, the development tools, hosting technology, and even system architectures they were built on can become increasingly obsolete. As new features and functionality are added, the complexity of these applications can increase dramatically, making them harder to maintain or add new features to.

Completely replacing a complex system can be a huge undertaking. Often, you will need a gradual migration to a new system, while keeping the old system to handle features that haven't been migrated yet. However, running two separate versions of an application means that clients have to know where particular features are located. Every time a feature or service is migrated, clients need to be updated to point to the new location.

Solution

Incrementally replace specific pieces of functionality with new applications and services. Create a façade that intercepts requests going to the backend legacy system. The façade routes these requests either to the legacy application or the new services. Existing features can be migrated to the new system gradually, and consumers can continue using the same interface, unaware that any migration has taken place.

This pattern helps to minimize risk from the migration, and spread the development effort over time. With the façade safely routing users to the correct application, you can add functionality to the new system at whatever pace you like, while ensuring the legacy application continues to function. Over time, as features are migrated to the new system, the legacy system is eventually "strangled" and is no longer necessary. Once this process is complete, the legacy system can safely be retired.

Issues and considerations

- Consider how to handle services and data stores that are potentially used by both new and legacy systems. Make sure both can access these resources side-by-side.
- Structure new applications and services in a way that they can easily be intercepted and replaced in future strangler fig migrations.
- At some point, when the migration is complete, the strangler fig façade will either go away or evolve into an adaptor for legacy clients.

- Make sure the façade keeps up with the migration.
- Make sure the façade doesn't become a single point of failure or a performance bottleneck.

When to use this pattern

Use this pattern when gradually migrating a back-end application to a new architecture.

This pattern may not be suitable:

- When requests to the back-end system cannot be intercepted.
- For smaller systems where the complexity of wholesale replacement is low.

Related guidance

- Martin Fowler's blog post on StranglerApplication

Throttling pattern

11/19/2021 • 7 minutes to read • Edit Online

Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service. This can allow the system to continue to function and meet service level agreements, even when an increase in demand places an extreme load on resources.

Context and problem

The load on a cloud application typically varies over time based on the number of active users or the types of activities they are performing. For example, more users are likely to be active during business hours, or the system might be required to perform computationally expensive analytics at the end of each month. There might also be sudden and unanticipated bursts in activity. If the processing requirements of the system exceed the capacity of the resources that are available, it'll suffer from poor performance and can even fail. If the system has to meet an agreed level of service, such failure could be unacceptable.

There're many strategies available for handling varying load in the cloud, depending on the business goals for the application. One strategy is to use autoscaling to match the provisioned resources to the user needs at any given time. This has the potential to consistently meet user demand, while optimizing running costs. However, while autoscaling can trigger the provisioning of additional resources, this provisioning isn't immediate. If demand grows quickly, there can be a window of time where there's a resource deficit.

Solution

An alternative strategy to autoscaling is to allow applications to use resources only up to a limit, and then throttle them when this limit is reached. The system should monitor how it's using resources so that, when usage exceeds the threshold, it can throttle requests from one or more users. This will enable the system to continue functioning and meet any service level agreements (SLAs) that are in place. For more information on monitoring resource usage, see the Instrumentation and Telemetry Guidance.

The system could implement several throttling strategies, including:

- Rejecting requests from an individual user who's already accessed system APIs more than n times per second over a given period of time. This requires the system to meter the use of resources for each tenant or user running an application. For more information, see the Service Metering Guidance.
- Disabling or degrading the functionality of selected nonessential services so that essential services can run unimpeded with sufficient resources. For example, if the application is streaming video output, it could switch to a lower resolution.
- Using load leveling to smooth the volume of activity (this approach is covered in more detail by the Queue-based Load Leveling pattern). In a multi-tenant environment, this approach will reduce the performance for every tenant. If the system must support a mix of tenants with different SLAs, the work for high-value tenants might be performed immediately. Requests for other tenants can be held back, and handled when the backlog has eased. The Priority Queue pattern could be used to help implement this approach.
- Deferring operations being performed on behalf of lower priority applications or tenants. These operations can be suspended or limited, with an exception generated to inform the tenant that the system is busy and that the operation should be retried later.

The figure shows an area graph for resource use (a combination of memory, CPU, bandwidth, and other factors)

against time for applications that are making use of three features. A feature is an area of functionality, such as a component that performs a specific set of tasks, a piece of code that performs a complex calculation, or an element that provides a service such as an in-memory cache. These features are labeled A, B, and C.

> The area immediately below the line for a feature indicates the resources that are used by applications when they invoke this feature. For example, the area below the line for Feature A shows the resources used by applications that are making use of Feature A, and the area between the lines for Feature A and Feature B indicates the resources used by applications invoking Feature B. Aggregating the areas for each feature shows the total resource use of the system.

The previous figure illustrates the effects of deferring operations. Just prior to time T1, the total resources allocated to all applications using these features reach a threshold (the limit of resource use). At this point, the applications are in danger of exhausting the resources available. In this system, Feature B is less critical than Feature A or Feature C, so it's temporarily disabled and the resources that it was using are released. Between times T1 and T2, the applications using Feature A and Feature C continue running as normal. Eventually, the resource use of these two features diminishes to the point when, at time T2, there is sufficient capacity to enable Feature B again.

The autoscaling and throttling approaches can also be combined to help keep the applications responsive and within SLAs. If the demand is expected to remain high, throttling provides a temporary solution while the system scales out. At this point, the full functionality of the system can be restored.

The next figure shows an area graph of the overall resource use by all applications running in a system against time, and illustrates how throttling can be combined with autoscaling.

At time T1, the threshold specifying the soft limit of resource use is reached. At this point, the system can start to scale out. However, if the new resources don't become available quickly enough, then the existing resources might be exhausted and the system could fail. To prevent this from occurring, the system is temporarily throttled, as described earlier. When autoscaling has completed and the additional resources are available, throttling can be relaxed.

Issues and considerations

You should consider the following points when deciding how to implement this pattern:

- Throttling an application, and the strategy to use, is an architectural decision that impacts the entire design of a system. Throttling should be considered early in the application design process because it isn't easy to add once a system has been implemented.
- Throttling must be performed quickly. The system must be capable of detecting an increase in activity and react accordingly. The system must also be able to revert to its original state quickly after the load has eased. This requires that the appropriate performance data is continually captured and monitored.
- If a service needs to temporarily deny a user request, it should return a specific error code so the client application understands that the reason for the refusal to perform an operation is due to throttling. The client application can wait for a period before retrying the request.
- Throttling can be used as a temporary measure while a system autoscales. In some cases it's better to simply throttle, rather than to scale, if a burst in activity is sudden and isn't expected to be long lived because scaling can add considerably to running costs.
- If throttling is being used as a temporary measure while a system autoscales, and if resource demands grow very quickly, the system might not be able to continue functioning—even when operating in a throttled mode. If this isn't acceptable, consider maintaining larger capacity reserves and configuring more aggressive autoscaling.

When to use this pattern

Use this pattern:

- To ensure that a system continues to meet service level agreements.
- To prevent a single tenant from monopolizing the resources provided by an application.
- To handle bursts in activity.
- To help cost-optimize a system by limiting the maximum resource levels needed to keep it functioning.

Example

The final figure illustrates how throttling can be implemented in a multi-tenant system. Users from each of the tenant organizations access a cloud-hosted application where they fill out and submit surveys. The application contains instrumentation that monitors the rate at which these users are submitting requests to the application.

In order to prevent the users from one tenant affecting the responsiveness and availability of the application for all other users, a limit is applied to the number of requests per second the users from any one tenant can submit. The application blocks requests that exceed this limit.

Next steps

The following guidance may also be relevant when implementing this pattern:

- Instrumentation and Telemetry Guidance. Throttling depends on gathering information about how heavily a service is being used. Describes how to generate and capture custom monitoring information.
- Service Metering Guidance. Describes how to meter the use of services in order to gain an understanding of how they are used. This information can be useful in determining how to throttle a service.
- Autoscaling Guidance. Throttling can be used as an interim measure while a system autoscales, or to remove the need for a system to autoscale. Contains information on autoscaling strategies.

Related guidance

The following patterns may also be relevant when implementing this pattern:

- Queue-based Load Leveling pattern. Queue-based load leveling is a commonly used mechanism for

implementing throttling. A queue can act as a buffer that helps to even out the rate at which requests sent by an application are delivered to a service.

- Priority Queue pattern. A system can use priority queuing as part of its throttling strategy to maintain performance for critical or higher value applications, while reducing the performance of less important applications.

Valet Key pattern

11/19/2021 • 12 minutes to read • Edit Online

Use a token that provides clients with restricted direct access to a specific resource, in order to offload data transfer from the application. This is particularly useful in applications that use cloud-hosted storage systems or queues, and can minimize cost and maximize scalability and performance.

Context and problem

Client programs and web browsers often need to read and write files or data streams to and from an application's storage. Typically, the application will handle the movement of the data — either by fetching it from storage and streaming it to the client, or by reading the uploaded stream from the client and storing it in the data store. However, this approach absorbs valuable resources such as compute, memory, and bandwidth.

Data stores have the ability to handle upload and download of data directly, without requiring that the application perform any processing to move this data. But, this typically requires the client to have access to the security credentials for the store. This can be a useful technique to minimize data transfer costs and the requirement to scale out the application, and to maximize performance. It means, though, that the application is no longer able to manage the security of the data. After the client has a connection to the data store for direct access, the application can't act as the gatekeeper. It's no longer in control of the process and can't prevent subsequent uploads or downloads from the data store.

This isn't a realistic approach in distributed systems that need to serve untrusted clients. Instead, applications must be able to securely control access to data in a granular way, but still reduce the load on the server by setting up this connection and then allowing the client to communicate directly with the data store to perform the required read or write operations.

Solution

You need to resolve the problem of controlling access to a data store where the store can't manage authentication and authorization of clients. One typical solution is to restrict access to the data store's public connection and provide the client with a key or token that the data store can validate.

This key or token is usually referred to as a valet key. It provides time-limited access to specific resources and allows only predefined operations such as reading and writing to storage or queues, or uploading and downloading in a web browser. Applications can create and issue valet keys to client devices and web browsers quickly and easily, allowing clients to perform the required operations without requiring the application to directly handle the data transfer. This removes the processing overhead, and the impact on performance and scalability, from the application and the server.

The client uses this token to access a specific resource in the data store for only a specific period, and with specific restrictions on access permissions, as shown in the figure. After the specified period, the key becomes invalid and won't allow access to the resource.

It's also possible to configure a key that has other dependencies, such as the scope of the data. For example, depending on the data store capabilities, the key can specify a complete table in a data store, or only specific rows in a table. In cloud storage systems the key can specify a container, or just a specific item within a container.

The key can also be invalidated by the application. This is a useful approach if the client notifies the server that the data transfer operation is complete. The server can then invalidate that key to prevent further access.

Using this pattern can simplify managing access to resources because there's no requirement to create and authenticate a user, grant permissions, and then remove the user again. It also makes it easy to limit the location, the permission, and the validity period—all by simply generating a key at runtime. The important factors are to limit the validity period, and especially the location of the resource, as tightly as possible so that the recipient can only use it for the intended purpose.

Issues and considerations

Consider the following points when deciding how to implement this pattern:

Manage the validity status and period of the key. If leaked or compromised, the key effectively unlocks the target item and makes it available for malicious use during the validity period. A key can usually be revoked or disabled, depending on how it was issued. Server-side policies can be changed or, the server key it was signed with can be invalidated. Specify a short validity period to minimize the risk of allowing unauthorized operations to take place against the data store. However, if the validity period is too short, the client might not be able to complete the operation before the key expires. Allow authorized users to renew the key before the validity period expires if multiple accesses to the protected resource are required.

Control the level of access the key will provide. Typically, the key should allow the user to only perform the actions necessary to complete the operation, such as read-only access if the client shouldn't be able to upload data to the data store. For file uploads, it's common to specify a key that provides write-only permission, as well as the location and the validity period. It's critical to accurately specify the resource or the set of resources to which the key applies.

Consider how to control users' behavior. Implementing this pattern means some loss of control over the resources users are granted access to. The level of control that can be exerted is limited by the capabilities of the policies and permissions available for the service or the target data store. For example, it's usually not possible to create a key that limits the size of the data to be written to storage, or the number of times the key can be used to access a file. This can result in huge unexpected costs for data transfer, even when used by the intended client, and might be caused by an error in the code that causes repeated upload or download. To limit the number of times a file can be uploaded, where possible, force the client to notify the application when one operation has completed. For example, some data stores raise events the application code can use to monitor operations and control user behavior. However, it's hard to enforce quotas for individual users in a multi-tenant scenario where the same key is used by all the users from one tenant.

Validate, and optionally sanitize, all uploaded data. A malicious user that gains access to the key could upload data designed to compromise the system. Alternatively, authorized users might upload data that's invalid and, when processed, could result in an error or system failure. To protect against this, ensure that all uploaded data is validated and checked for malicious content before use.

Audit all operations. Many key-based mechanisms can log operations such as uploads, downloads, and failures. These logs can usually be incorporated into an audit process, and also used for billing if the user is charged based on file size or data volume. Use the logs to detect authentication failures that might be caused by issues with the key provider, or accidental removal of a stored access policy.

Deliver the key securely. It can be embedded in a URL that the user activates in a web page, or it can be used in a server redirection operation so that the download occurs automatically. Always use HTTPS to deliver the key over a secure channel.

Protect sensitive data in transit. Sensitive data delivered through the application will usually take place using SSL or TLS, and this should be enforced for clients accessing the data store directly.

Other issues to be aware of when implementing this pattern are:

- If the client doesn't, or can't, notify the server of completion of the operation, and the only limit is the expiration period of the key, the application won't be able to perform auditing operations such as counting the number of uploads or downloads, or preventing multiple uploads or downloads.
- The flexibility of key policies that can be generated might be limited. For example, some mechanisms only allow the use of a timed expiration period. Others aren't able to specify a sufficient granularity of read/write permissions.
- If the start time for the key or token validity period is specified, ensure that it's a little earlier than the current server time to allow for client clocks that might be slightly out of synchronization. The default, if not specified, is usually the current server time.
- The URL containing the key will be recorded in server log files. While the key will typically have expired before the log files are used for analysis, ensure that you limit access to them. If log data is transmitted to a monitoring system or stored in another location, consider implementing a delay to prevent leakage of keys until after their validity period has expired.
- If the client code runs in a web browser, the browser might need to support cross-origin resource sharing (CORS) to enable code that executes within the web browser to access data in a different domain from the one that served the page. Some older browsers and some data stores don't support CORS, and code that runs in these browsers might not be able to use a valet key to provide access to data in a different domain, such as a cloud storage account.

When to use this pattern

This pattern is useful for the following situations:

- To minimize resource loading and maximize performance and scalability. Using a valet key doesn't require the resource to be locked, no remote server call is required, there's no limit on the number of valet keys that can be issued, and it avoids a single point of failure resulting from performing the data transfer through the application code. Creating a valet key is typically a simple cryptographic operation of signing a string with a key.
- To minimize operational cost. Enabling direct access to stores and queues is resource and cost efficient, can result in fewer network round trips, and might allow for a reduction in the number of compute resources required.
- When clients regularly upload or download data, particularly where there's a large volume or when each

operation involves large files.

- When the application has limited compute resources available, either due to hosting limitations or cost considerations. In this scenario, the pattern is even more helpful if there are many concurrent data uploads or downloads because it relieves the application from handling the data transfer.
- When the data is stored in a remote data store or a different datacenter. If the application was required to act as a gatekeeper, there might be a charge for the additional bandwidth of transferring the data between datacenters, or across public or private networks between the client and the application, and then between the application and the data store.

This pattern might not be useful in the following situations:

- If the application must perform some task on the data before it's stored or before it's sent to the client. For example, if the application needs to perform validation, log access success, or execute a transformation on the data. However, some data stores and clients are able to negotiate and carry out simple transformations such as compression and decompression (for example, a web browser can usually handle gzip formats).
- If the design of an existing application makes it difficult to incorporate the pattern. Using this pattern typically requires a different architectural approach for delivering and receiving data.
- If it's necessary to maintain audit trails or control the number of times a data transfer operation is executed, and the valet key mechanism in use doesn't support notifications that the server can use to manage these operations.
- If it's necessary to limit the size of the data, especially during upload operations. The only solution to this is for the application to check the data size after the operation is complete, or check the size of uploads after a specified period or on a scheduled basis.

Example

Azure supports shared access signatures on Azure Storage for granular access control to data in blobs, tables, and queues, and for Service Bus queues and topics. A shared access signature token can be configured to provide specific access rights such as read, write, update, and delete to a specific table; a key range within a table; a queue; a blob; or a blob container. The validity can be a specified time period or with no time limit.

Azure shared access signatures also support server-stored access policies that can be associated with a specific resource such as a table or blob. This feature provides additional control and flexibility compared to application-generated shared access signature tokens, and should be used whenever possible. Settings defined in a server-stored policy can be changed and are reflected in the token without requiring a new token to be issued, but settings defined in the token can't be changed without issuing a new token. This approach also makes it possible to revoke a valid shared access signature token before it's expired.

> For more information, see Grant limited access to Azure Storage resources using shared access signatures (SAS).

The following code shows how to create a shared access signature token that's valid for five minutes. The `GetSharedAccessReferenceForUpload` method returns a shared access signatures token that can be used to upload a file to Azure Blob Storage.

```
public class ValuesController : ApiController
{
  private readonly BlobServiceClient blobServiceClient;
  private readonly string blobContainer;
  ...
  /// <summary>
  /// Return a limited access key that allows the caller to upload a file
  /// to this specific destination for a defined period of time.
  /// </summary>
  private StorageEntitySas GetSharedAccessReferenceForUpload(string blobName)
  {
      var blob = blobServiceClient.GetBlobContainerClient(this.blobContainer).GetBlobClient(blobName);
      var storageSharedKeyCredential = new StorageSharedKeyCredential(blobServiceClient.AccountName,
ConfigurationManager.AppSettings["AzureStorageEmulatorAccountKey"]);

      var blobSasBuilder = new BlobSasBuilder
      {
          BlobContainerName = this.blobContainer,
          BlobName = blobName,
          Resource = "b",
          StartsOn = DateTimeOffset.UtcNow.AddMinutes(-5),
          ExpiresOn = DateTimeOffset.UtcNow.AddMinutes(5)
      };
      blobSasBuilder.SetPermissions(BlobSasPermissions.Write);

      return new StorageEntitySas
      {
          BlobUri = blob.Uri,
          Credentials = blobSasBuilder.ToSasQueryParameters(storageSharedKeyCredential).ToString()
      };
  }
  public struct StorageEntitySas
  {
      public string Credentials;
      public Uri BlobUri;
  }
}
```

> The complete sample is available in the ValetKey solution available for download from GitHub. The ValetKey.Web project in this solution contains a web application that includes the `ValuesController` class shown above. A sample client application that uses this web application to retrieve a shared access signatures key and upload a file to blob storage is available in the ValetKey.Client project.

Next steps

The following guidance might be relevant when implementing this pattern:

- A sample that demonstrates this pattern is available on GitHub.
- Grant limited access to Azure Storage resources using shared access signatures (SAS)
- Shared Access Signature Authentication with Service Bus

Related guidance

The following patterns might also be relevant when implementing this pattern:

- Gatekeeper pattern. This pattern can be used in conjunction with the Valet Key pattern to protect applications and services by using a dedicated host instance that acts as a broker between clients and the application or service. The gatekeeper validates and sanitizes requests, and passes requests and data between the client and the application. Can provide an additional layer of security, and reduce the attack surface of the system.
- Static Content Hosting pattern. Describes how to deploy static resources to a cloud-based storage service

that can deliver these resources directly to the client to reduce the requirement for expensive compute instances. Where the resources aren't intended to be publicly available, the Valet Key pattern can be used to secure them.

Microsoft Azure Well-Architected Framework

11/19/2021 • 4 minutes to read • Edit Online

The Azure Well-Architected Framework is a set of guiding tenets that can be used to improve the quality of a workload. The framework consists of five pillars of architectural excellence:

- Reliability
- Security
- Cost Optimization
- Operational Excellence
- Performance Efficiency

Incorporating these pillars helps produce a high quality, stable, and efficient cloud architecture:

PILLAR	DESCRIPTION
Reliability	The ability of a system to recover from failures and continue to function.
Security	Protecting applications and data from threats.
Cost Optimization	Managing costs to maximize the value delivered.
Operational Excellence	Operations processes that keep a system running in production.
Performance Efficiency	The ability of a system to adapt to changes in load.

Reference the following video about how to architect successful workloads on Azure with the Well-Architected Framework:

https://channel9.msdn.com/Shows/Azure-Enablement/Architect-successful-workloads-on-Azure--Introduction-Ep-1-Well-Architected-series/player

Overview

The following diagram gives a high-level overview of the Azure Well-Architected Framework:

In the center, is the Well-Architected Framework, which includes the five pillars of architectural excellence. Surrounding the Well-Architected Framework are six supporting elements:

- Azure Well-Architected Review
- Azure Advisor
- Documentation
- Partners, Support, and Services Offers
- Reference Architectures
- Design Principles

Assess your workload

To assess your workload using the tenets found in the Microsoft Azure Well-Architected Framework, see the Microsoft Azure Well-Architected Review.

We also recommend you use Azure Advisor and Advisor Score to identify and prioritize opportunities to improve the posture of your workloads. Both services are free to all Azure users and align to the five pillars of

the Well-Architected Framework:

- **Azure Advisor** is a personalized cloud consultant that helps you follow best practices to optimize your Azure deployments. It analyzes your resource configuration and usage telemetry. It recommends solutions that can help you improve the reliability, security, cost effectiveness, performance, and operational excellence of your Azure resources. Learn more about Azure Advisor.
- **Advisor Score** is a core feature of Azure Advisor that aggregates Advisor recommendations into a simple, actionable score. This score enables you to tell at a glance if you're taking the necessary steps to build reliable, secure, and cost-efficient solutions, and to prioritize the actions that will yield the biggest improvement to the posture of your workloads. The Advisor score consists of an overall score, which can be further broken down into five category scores corresponding to each of the Well-Architected pillars. Learn more about Advisor Score.

Reliability

A reliable workload is one that is both resilient and available. Resiliency is the ability of the system to recover from failures and continue to function. The goal of resiliency is to return the application to a fully functioning state after a failure occurs. Availability is whether your users can access your workload when they need to.

For more information about resiliency, reference the following video that will show you how to start improving the reliability of your Azure workloads:

https://channel9.msdn.com/Shows/Azure-Enablement/Start-improving-the-reliability-of-your-Azure-workloads--Reliability-Ep-1--Well-Architected-series/player

Reliability guidance

The following topics offer guidance on designing and improving reliable Azure applications:

- Designing reliable Azure applications
- Design patterns for resiliency
- Best practices:
  - Transient fault handling
  - Retry guidance for specific services

For an overview of reliability principles, reference Principles of the reliability pillar.

Security

Think about security throughout the entire lifecycle of an application, from design and implementation to deployment and operations. The Azure platform provides protections against various threats, such as network intrusion and DDoS attacks. But you still need to build security into your application and into your DevOps processes.

Ask the right questions about secure application development on Azure by referencing the following video:

https://channel9.msdn.com/Shows/Azure-Enablement/Ask-the-right-questions-about-secure-application-development-on-Azure/player

Security guidance

Consider the following broad security areas:

- Identity management
- Protect your infrastructure
- Application security
- Data sovereignty and encryption

- Security resources

For more information, reference Overview of the security pillar.

Cost optimization

When you're designing a cloud solution, focus on generating incremental value early. Apply the principles of **Build-Measure-Learn**, to accelerate your time to market while avoiding capital-intensive solutions.

For more information, reference Cost optimization and the following video on how to start optimizing your Azure costs:

https://channel9.msdn.com/Shows/Azure-Enablement/Start-optimizing-your-Azure-costs--Cost-Optimization-Ep-1--Well-Architected-series/player

Cost guidance

The following topics offer cost optimization guidance as you develop the Well-Architected Framework for your workload:

- Review cost principles
- Develop a cost model
- Create budgets and alerts
- Review the cost optimization checklist

For a high-level overview, reference Overview of the cost optimization pillar.

Operational excellence

Operational excellence covers the operations and processes that keep an application running in production. Deployments must be reliable and predictable. Automate deployments to reduce the chance of human error. Fast and routine deployment processes won't slow down the release of new features or bug fixes. Equally important, you must quickly roll back or roll forward if an update has problems.

For more information, reference the following video about bringing security into your DevOps practice on Azure:

https://channel9.msdn.com/Shows/Azure-Enablement/DevSecOps-bringing-security-into-your-DevOps-practice-on-Azure/player

Operational excellence guidance

The following topics provide guidance on designing and implementing DevOps practices for your Azure workload:

- Design patterns for operational excellence
- Best practices: Monitoring and diagnostics

For a high-level summary, reference Overview of the operational excellence pillar.

Performance efficiency

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. The main ways to achieve performance efficiency include using scaling appropriately and implementing PaaS offerings that have scaling built in.

For more information, tune in to Performance Efficiency: Fast & Furious: Optimizing for Quick & Reliable VM Deployments:

Performance efficiency guidance

The following topics offer guidance on how to design and improve the performance efficiency posture of your Azure workload:

- Design patterns for performance efficiency
- Best practices:
  - Autoscaling
  - Background jobs
  - Caching
  - CDN
  - Data partitioning

For a high-level synopsis, reference Overview of the performance efficiency pillar.

Next steps

Learn more about:

- Azure Well-Architected Review
- Well-Architected Series
- Introduction to the Microsoft Azure Well-Architected Framework
- Microsoft Defender for Cloud
- Cloud Adoption Framework

Overview of the reliability pillar

11/19/2021 • 4 minutes to read • Edit Online

Reliability ensures your application can meet the commitments you make to your customers. Architecting resiliency into your application framework ensures your workloads are available and can recover from failures at any scale.

Building for reliability includes:

- Ensuring a highly available architecture
- Recovering from failures such as data loss, major downtime, or ransomware incidents

To assess the reliability of your workload using the tenets found in the Microsoft Azure Well-Architected Framework, reference the Microsoft Azure Well-Architected Review.

For more information, explore the following video on diving deeper into Azure workload reliability:

In traditional application development, there has been a focus on increasing the mean time between failures (MTBF). Effort was spent trying to prevent the system from failing. In cloud computing, a different mindset is required, because of several factors:

- Distributed systems are complex, and a failure at one point can potentially cascade throughout the system.
- Costs for cloud environments are kept low through commodity hardware, so occasional hardware failures must be expected.
- Applications often depend on external services, which may become temporarily unavailable or throttle high-volume users.
- Today's users expect an application to be available 24/7 without ever going offline.

All of these factors mean that cloud applications must be designed to expect occasional failures and recover from them. Azure has many resiliency features already built into the platform. For example:

- Azure Storage, SQL Database, and Cosmos DB all provide built-in data replication across availability zones and regions.
- Azure managed disks are automatically placed in different storage scale units to limit the effects of hardware failures.
- Virtual machines (VMs) in an availability set are spread across several fault domains. A *fault domain* is a group of VMs that share a common power source and network switch. Spreading VMs across fault domains limits the impact of physical hardware failures, network outages, or power interruptions.
- *Availability Zones* are physically separate locations within each Azure region. Each zone is composed of one or more datacenters equipped with independent power, cooling, and networking infrastructure. With availability zones, you can design and operate applications, and databases that automatically transition between zones without interruption, which ensures resiliency if one zone is affected. For more information, reference Regions and Availability Zones in Azure.

That said, you still need to build resiliency into your application. Resiliency strategies can be applied at all levels of the architecture. Some mitigations are more tactical in nature—for example, retrying a remote call after a transient network failure. Other mitigations are more strategic, such as failing over the entire application to a secondary region. Tactical mitigations can make a large difference. While it's rare for an entire region to experience a disruption, transient problems such as network congestion are more common—so target these

issues first. Having the right monitoring and diagnostics is also important, both to detect failures when they happen, and to find the root causes.

When designing an application to be resilient, you must understand your availability requirements. How much downtime is acceptable? The amount of downtime is partly a function of cost. How much will potential downtime cost your business? How much should you invest in making the application highly available?

Topics and best practices

The reliability pillar covers the following topics and best practices to help you build a resilient workload:

RELIABILITY TOPIC	DESCRIPTION
Reliability principles	These critical principles are used as lenses to assess the reliability of an application deployed on Azure.
Design for reliability	Consider how systems use Availability Zones, perform scalability, respond to failure, and other strategies that optimize reliability in application design.
Resiliency checklist for specific Azure services	Every technology has its own particular failure modes, which you must consider when designing and implementing your application. Use this checklist to review the resiliency considerations for specific Azure services.
Target and non-functional requirements	Target and non-functional requirements such as availability targets and recovery targets allow you to measure the uptime and downtime of your workloads. Having clearly defined targets is crucial to have a goal to work and measure against.
Resiliency and dependencies	Building failure recovery into the system should be part of the architecture and design phases from the beginning to avoid the risk of failure. Dependencies are required for the application to fully operate.
Availability Zones	Availability Zones can be used to spread a solution across multiple zones within a region, allowing for an application to continue functioning when one zone fails.
Availability of services	Availability of services across Azure regions depends on a region's type. Azure's general policy on deploying services into any given region is primarily driven by region type, service categories, and customer demand.
Availability zone terminology	To better understand regions and availability zones in Azure, it helps to understand key terms or concepts.
Best practices	During the architectural phase, focus on implementing practices that meet your business requirements, identify failure points, and minimize the scope of failures.
Testing for reliability	Regular testing should be performed as part of each major change to validate existing thresholds, targets, and assumptions.

RELIABILITY TOPIC	DESCRIPTION
Monitoring for reliability	Get an overall picture of application health. If something fails, you need to know *that* it failed, *when* it failed, and *why*.
Reliability patterns	Applications must be designed and implemented to maximize availability.

Next step

Principles

Principles of the reliability pillar

11/19/2021 • 2 minutes to read • Edit Online

Building a reliable application in the cloud is different from traditional application development. While historically you may have purchased levels of redundant higher-end hardware to minimize the chance of an entire application platform failing, in the cloud, we acknowledge up front that failures will happen. Instead of trying to prevent failures altogether, the goal is to minimize the effects of a single failing component.

Application framework

These critical principles are used as lenses to assess the reliability of an application deployed on Azure. They provide a framework for the application assessment questions that follow.

To assess your workload using the tenets found in the Microsoft Azure Well-Architected Framework, see the Microsoft Azure Well-Architected Review.

- **Define and test availability and recovery targets:** Availability targets, such as Service Level Agreements (SLA) and Service Level Objectives (SLO), and Recovery targets, such as Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO), should be defined and tested to ensure application reliability aligns with business requirements.
- **Design applications to be resistant to failures:** Resilient application architectures should be designed to recover gracefully from failures in alignment with defined reliability targets.
- **Ensure required capacity and services are available in targeted regions:** Azure services and capacity can vary by region, so it's important to understand if targeted regions offer required capabilities.
- **Plan for disaster recovery:** Disaster recovery is the process of restoring application functionality in the wake of a catastrophic failure. It might be acceptable for some applications to be unavailable or partially available with reduced functionality for a period of time, while other applications may not be able to tolerate reduced functionality.
- **Design the application platform to meet reliability requirements:** Designing application platform resiliency and availability is critical to ensuring overall application reliability.
- **Design the data platform to meet reliability requirements:** Designing data platform resiliency and availability is critical to ensuring overall application reliability.
- **Recover from errors:** Resilient applications should be able to automatically recover from errors by leveraging modern cloud application code patterns.
- **Ensure networking and connectivity meets reliability requirements:** Identifying and mitigating potential network bottle-necks or points-of-failure supports a reliable and scalable foundation over which resilient application components can communicate.
- **Allow for reliability in scalability and performance:** Resilient applications should be able to automatically scale in response to changing load to maintain application availability and meet performance requirements.
- **Address security-related risks:** Identifying and addressing security-related risks helps to minimize application downtime and data loss caused by unexpected security exposures.
- **Define, automate, and test operational processes:** Operational processes for application deployment, such as roll-forward and roll-back, should be defined, sufficiently automated, and tested to

help ensure alignment with reliability targets.

- **Test for fault tolerance:** Application workloads should be tested to validate reliability against defined reliability targets.
- **Monitor and measure application health:** Monitoring and measuring application availability is vital to qualifying overall application health and progress towards defined reliability targets.

Next step

Design

Design for reliability

11/19/2021 • 2 minutes to read • Edit Online

Reliable applications should maintain a pre-defined percentage of uptime (*availability*). They should also balance between high resiliency, low latency, and cost (*High Availability*). Just as important, applications should be able to recover from failures (*resiliency*).

Checklist

How have you designed your applications with reliability in mind?

- Define availability and recovery targets to meet business requirements.
- Build resiliency and availability into your apps by gathering requirements.
- Ensure that application and data platforms meet your reliability requirements.
- Configure connection paths to promote availability.
- Use Availability Zones where applicable to improve reliability and optimize costs.
- Ensure that your application architecture is resilient to failures.
- Know what happens if the requirements of Service Level Agreements are not met.
- Identify possible failure points in the system to build resiliency.
- Ensure that applications can operate in the absence of their dependencies.

Azure services

- Azure Front Door
- Azure Traffic Manager
- Azure Load Balancer
- Service Fabric
- Kubernetes Service (AKS)
- Azure Site Recovery

Reference architecture

- Deploy highly available network virtual appliances
- Failure Mode Analysis for Azure applications
- Minimize coordination

Next step

Target & non-functional requirements

Related links

- Use platform as a service (PaaS) options
- Design to scale out
- Workload availability targets.
- Building solutions for high availability using Availability Zones
- Make all things redundant

Target and non-functional requirements

11/19/2021 • 12 minutes to read • Edit Online

Target and non-functional requirements such as *availability targets* and *recovery targets* allow you to measure the uptime and downtime of your workloads. Having clearly defined targets is crucial in order to have a goal to work and measure against. In addition to these targets, there are many other requirements you should consider to improve reliability requirements and meet business expectations.

Building resiliency (recovering from failures) and availability (running in a healthy state without significant downtime) into your apps begins with gathering requirements. For example, how much downtime is acceptable? How much does potential downtime cost your business? What are your customer's availability requirements? How much do you invest in making your application highly available? What is the risk versus the cost?

Key points

- Determine the acceptable level of uptime for your workloads.
- Determine how long workloads can be unavailable and how much data is acceptable to lose during a disaster.
- Consider application and data platform requirements to improve resiliency and availability.
- Ensure connection availability and improve reliability with Azure services.
- Assess overall application health of workloads.

Availability targets

A Service Level Agreement (SLA), is an availability target that represents a commitment around performance and availability of the application. Understanding the SLA of individual components within the system is essential in order to define reliability targets. Knowing the SLA of dependencies will also provide a justification for additional spend when making the dependencies highly available and with proper support contracts. Availability targets for any dependencies leveraged by the application should be understood and ideally align with application targets should also be considered.

Understanding your availability expectations is vital to reviewing overall operations for the application. For example, if you are striving to achieve an application Service Level Objective (SLO) of 99.999%, the level of inherent operational action required by the application is going to be far greater than if an SLO of 99.9% was the goal.

Monitoring and measuring application availability is vital to qualifying overall application health and progress towards defined targets. Make sure you measure and monitor key targets such as:

- Mean Time Between Failures (MTBF) — The average time between failures of a particular component.
- Mean Time To Recover (MTTR) — The average time it takes to restore a component after a failure.

Considerations for availability targets

Are SLAs/SLOs/SLIs for all leveraged dependencies understood?

Availability targets for any dependencies leveraged by the application should be understood and ideally align with application targets. Make sure SLAs/SLOs/SLIs for all leveraged dependencies are understood.

Has a composite SLA been calculated for the application and/or key scenarios using Azure SLAs?

A composite SLA captures the end-to-end SLA across all application components and dependencies. It is calculated using the individual SLAs of Azure services housing application components and provides an important indicator of designed availability in relation to customer expectations and targets. Make sure the composite SLA of all components and dependencies on the critical paths are understood. To learn more, see Composite SLAs.

> **NOTE**
> if you have contractual commitments to an SLA for your Azure solution, additional allowances on top of the Azure composite SLA must be made to accommodate outages caused by code-level issues and deployments. This is often overlooked and customers directly put the composite SLA forward to their customers.

Are availability targets considered while the system is running in disaster recovery mode?

Availability targets might or might not be applied when running in disaster recovery mode. This depends from application to application. If targets must also apply in a failure state then an N+1 model should be used to achieve greater availability and resiliency. In this scenario, N is the capacity needed to deliver required availability. There's also a cost implication, because more resilient infrastructure usually is more expensive. This has to be accepted by business.

What are the consequences if availability targets are not satisfied?

Are there any penalties, such as financial charges, associated with failing to meet SLA commitments? Additional measures can be used to prevent penalties, but that also brings additional cost to operate the infrastructure. This has to be factored in and evaluated. It should be fully understood what are the consequences if availability targets are not satisfied. This will also inform when to initiate a failover case.

Recovery targets

Recovery targets identify how long the workload can be unavailable and how much data is acceptable to lose during a disaster. Define target reports for the application and key scenarios. Target reports needed are Recovery Time Objective (RTO) — the maximum acceptable time an application is unavailable after an incident, and Recovery Point Objective (RPO) — the maximum duration of data loss that is acceptable during a disaster.

Recovery targets are nonfunctional requirements of a system and should be dictated by business requirements. Recovery targets should be defined in accordance to the required RTO and RPO targets for the workloads.

Meet application platform requirements

Azure application platform services offer resiliency features to support application reliability, though they may only be applicable at a certain SKU and configuration/deployment. For example, an SLA is dependent on the number of instances deployed or a certain feature enabled. It is recommended that you review the SLA for services used. For example, Service Bus Premium SKU provides predictable latency and throughput to mitigate noisy neighbor scenarios. It also provides the ability to automatically scale and replicate metadata to another Service Bus instance for failover purposes.

To learn more, see Azure Service Bus Premium SKU.

Multiple and paired regions

An application platform should be deployed across multiple regions if the requirements dictate. Covering the requirements using zones is cheaper and less complex. Regional isolation should be an extra measure if the SLAs given by the single region cross-zone setup are insufficient or if required by a geographical spread of users.

The ability to respond to disaster scenarios for overall compute platform availability and application resiliency depends on the use of multiple regions or other deployment locations.

Use paired regions that exist within the same geography and provide native replication features for recovery purposes, such as Geo-Redundant Storage (GRS) asynchronous replication. In the event of planned maintenance, updates to a region will be performed sequentially only. To learn more, see Business continuity with Azure Paired Regions.

Availability Zones and sets

Platform services that can leverage Availability Zones are deployed in either a zonal manner within a particular zone, or in a zone-redundant configuration across multiple zones. To learn more, see Building solutions for high availability using Availability Zones.

An Availability Set (AS) is a logical construct to inform Azure that it should distribute contained virtual machine instances across multiple fault and update domains within an Azure region. Availability Zones (AZ) elevate the fault level for virtual machines to a physical datacenter by allowing replica instances to be deployed across multiple datacenters within an Azure region. While zones provide greater resiliency than sets, there are performance and cost considerations where applications are extremely 'chatty' across zones given the implied physical separation and inter-zone bandwidth charges. Ultimately, Azure Virtual Machines and Azure PaaS services, such as Service Fabric and Azure Kubernetes Service (AKS) which use virtual machines underneath, can leverage either AZs or an AS to provide application resiliency within a region. To learn more, see Business continuity with data resiliency.

Considerations for availability

Is the application hosted across 2 or more application platform nodes?

To ensure application platform reliability, it is vital that the application be hosted across at least two nodes to ensure there are no single points of failure. Ideally An n+1 model should be applied for compute availability where n is the number of instances required to support application availability and performance requirements.

> **NOTE**
>
> Higher SLAs provided for virtual machines and associated related platform services, require at least two replica nodes deployed to either an Availability Set or across two or more Availability Zones. To learn more, see SLA for Virtual Machines.

How is the client traffic routed to the application in the case of region, zone or network outage?

In the event of a major outage, client traffic should be routable to application deployments which remain available across other regions or zones. This is ultimately where cross-premises connectivity and global load balancing should be used, depending on whether the application is internal and/or external facing. Services such as Azure Front Door, Azure Traffic Manager, or third-party CDNs can route traffic across regions based on application health solicited via health probes. To learn more, see Traffic Manager endpoint monitoring.

Meet data platform requirements

Data and storage services should be running in a highly available configuration/SKU. Azure data platform services offer resiliency features to support application reliability, though they may only be applicable at a certain SKU. Examples are Azure SQL Database Business Critical SKUs, or Azure Storage Zone Redundant Storage (ZRS) with three synchronous replicas spread across availability zones.

Data consistency

Data types should be categorized by data consistency requirements. Data consistency requirements, such as strong or eventual consistency, should be understood for all data types and used to inform data grouping and

categorization, as well as what data replication/synchronization strategies can be considered to meet application reliability targets.

CAP theorem proves that it is impossible for a distributed data store to simultaneously provide more than two guarantees across:

- **Consistency:** Every read receives the most recent write or an error.
- **Availability:** Every request receives a non-error response, without the guarantee that it contains the most recent write.
- **Partition tolerance:** A system continues to operate despite an arbitrary number of transactions being dropped or delayed by the network between nodes.

Determining which of these guarantees are most important in the context of application requirements is critical.

Replication and Redundancy

Replicating data across zones or paired regions supports application availability objectives to limit the impact of failure scenarios. The ability to restore data from a backup is essential when recovering from data corruption situations as well as failure scenarios. To ensure sufficient redundancy and availability for zonal and regional failure scenarios, backups should be stored across zones and/or regions.

Define and test a data restore process to ensure a consistent application state. Regular testing of the data restore process promotes operational excellence and confidence in the ability to recover data in alignment with defined recovery objectives for the application.

Consider how your application traffic is routed to data sources in the case of region, zone, or network outage. Understanding the method used to route application traffic to data sources in the event of a major failure event is critical to identify whether failover processes will meet recovery objectives. Many Azure data platform services offer native reliability capabilities to handle major failures, such as Cosmos DB Automatic Failover or Azure SQL DB Active Geo-Replication.

> **NOTE**
> Some capabilities such as Azure Storage RA-GRS and Azure SQL DB Active Geo-Replication require application-side failover to alternate endpoints in some failure scenarios, so application logic should be developed to handle these scenarios.

Networking and connectivity requirements

Consider these guidelines to ensure connection availability and improve reliability with Azure services.

Connectivity

- **Use a global load balancer used to distribute traffic and/or failover across regions.** Azure Front Door, Azure Traffic Manager, or third-party CDN services can be used to direct inbound requests to external-facing application endpoints deployed across multiple regions. It is important to note that Traffic Manager is a DNS-based load balancer, so failover must wait for DNS propagation to occur. A sufficiently low TTL (Time To Live) value should be used for DNS records, though not all ISPs may honor this. For application scenarios requiring transparent failover, Azure Front Door should be used. To learn more, see Disaster Recovery using Azure Traffic Manager and Azure Front Door routing architecture.

- **For cross-premises connectivity (ExpressRoute or VPN) ensure there redundant connections from different locations.** At least two redundant connections should be established across two or more Azure regions and peering locations to ensure there are no single points of failure. An active/active load-shared configuration provides path diversity and promotes availability of network connection paths. To learn more, see Cross-network connectivity.

- **Simulate a failure path to ensure connectivity is available over alternative paths.** The failure of a connection path onto other connection paths should be tested to validate connectivity and operational effectiveness. Using Site-to-Site VPN connectivity as a backup path for ExpressRoute provides an additional layer of network resiliency for cross-premises connectivity. To learn more, see Using site-to-site VPN as a backup for ExpressRoute private peering.

- **Eliminate all single points of failure from the data path (on-premises and Azure.** Single-instance Network Virtual Appliances (NVAs), whether deployed in Azure or within an on-premises datacenter, introduce significant connectivity risk. To learn more, see Deploy highly available network virtual appliances.

Zone-aware services

- **Use ExpressRoute/VPN zone-redundant Virtual Network Gateways.** Zone-redundant virtual network gateways distribute gateway instances across Availability Zones to improve reliability and ensure availability during failure scenarios impacting a datacenter within a region. To learn more, see Zone-redundant Virtual Network Gateways.

- **If used, deploy Azure Application Gateway v2 deployed in a zone-redundant configuration.** Azure Application Gateway v2 can be deployed in a zone-redundant configuration to deploy gateway instances across zones for improved reliability and availability during failure scenarios impacting a datacenter within a region. To learn more, see Zone-redundant Application Gateway v2.

- **Use Azure Load Balancer Standard to load-balance traffic across Availability Zones.** Azure Load Balancer Standard is zone-aware to distribute traffic across Availability Zones. It can also be configured in a zone-redundant configuration to improve reliability and ensure availability during failure scenarios impacting a datacenter within a region. To learn more, see Standard Load Balancer and Availability Zones.

- **Configure health probes for Azure Load Balancer(s)/Azure Application Gateways.** Health probes allow Azure Load Balancers to assess the health of backend endpoints to prevent traffic from being sent to unhealthy instances. To learn more, see Load Balancer health probes.

- **Assess critical application dependencies with health probes.** Custom health probes should be used to assess overall application health including downstream components and dependent services, such as APIs and datastores, so that traffic is not sent to backend instances that cannot successfully process requests due to dependency failures. To learn more, see Health Endpoint Monitoring Pattern.

Next step

Application design

Related links

- To understand business metrics to design resilient Azure applications, see Workload availability targets.
- For information on Availability Zones, see Building solutions for high availability using Availability Zones.
- For information on health probes, see Load Balancer health probes and Health Endpoint Monitoring Pattern.
- To learn about connectivity risk, see Deploy highly available network virtual appliances.

Go back to the main article: Design

Design reliable Azure applications

11/19/2021 • 4 minutes to read • Edit Online

Building a reliable application in the cloud is different from traditional application development. While historically you may have purchased levels of redundant higher-end hardware to minimize the chance of an entire application platform failing, in the cloud, we acknowledge up front that failures will happen. Instead of trying to prevent failures altogether, the goal is to minimize the effects of a single failing component. Failures you can expect here are inherent to highly distributed systems, not a feature of Azure.

Key Points

- Use Availability Zones where applicable to improve reliability and optimize costs.
- Design applications to operate when impacted by failures.
- Use the native resiliency capabilities of PaaS to support overall application reliability.
- Design to scale out.
- Validate that required capacity is within Azure service scale limits and quotas.

Use Availability Zones within a region

If your requirements demand an even greater failure isolation than Availability Zones alone can offer, consider deploying to multiple regions. Multiple regions should be used for failover purposes in a disaster state. Additional cost needs to be taken into consideration. Examples of cost needs are data and networking, and services such as Azure Site Recovery.

Design your application architecture to use *Availability Zones* within a region. Availability Zones can be used to optimize application availability within a region by providing datacenter level fault tolerance. However, the application architecture must not share dependencies between zones to use them effectively.

> **NOTE**
>
> Availability Zones may introduce performance and cost considerations for applications which are extremely "chatty" across zones given the implied physical separation between each zone and inter-zone bandwidth charges. This also means that Availability Zones can be considered to get higher SLA for lower cost.

Consider if component proximity is required for application performance reasons. If all or part of the application is highly sensitive to latency, it may mandate component co-locality which can limit the applicability of multi-region and multi-zone strategies.

Respond to failure

Avoiding failure is impossible in the public cloud, and as a result applications require resilience to respond to outages and deliver reliability. The application should therefore be designed to operate even when impacted by regional, zonal, service or component failures across critical application scenarios and functionality. Application operations may experience reduced functionality or degraded performance during an outage.

Define an availability strategy to capture how the application remains available when in a failure state. It should apply across all application components and the application deployment stamp as a whole such as via multi-geo scale-unit deployment approach. There are cost implications as well: More resources need to be provisioned in advance to provide high availability. Active-active setup, while more expensive than single deployment, can

balance cost by lowering load on one stamp and reducing the total amount of resources needed.

In addition to an availability strategy, define a Business Continuity Disaster Recovery (BCDR) strategy for the application and/or its key scenarios. A disaster recovery strategy should capture how the application responds to a disaster situation such as a regional outage or the loss of a critical platform service, using either a re-deployment, warm-spare active-passive, or hot-spare active-active approach.

To drive cost down consider splitting application components and data into groups. For example:

- Must protect
- Nice to protect
- Ephemeral/can be rebuilt/lost, instead of protecting all data with the same policy

Considerations for improving reliability

Is the application designed to use managed services?

Azure-managed services provide native resiliency capabilities to support overall application reliability. Platform as a service (PaaS) offerings should be used to leverage these capabilities. PaaS options are easier to configure and administer. You don't need to provision VMs, set up VNets, manage patches and updates, and all of the other overhead associated with running software on a VM. To learn more, see Use managed services.

Has the application been designed to scale out?

Azure provides elastic scalability and you should design to scale out. However, applications must leverage a scale-unit approach to navigate service and subscription limits to ensure that individual components and the application as a whole can scale horizontally. Don't forget about scale in, which is important to drive cost down. For example, scale in and out for App Service is done via rules. Often customers write scale out rules and never write scale in rules. This leaves the App Service more expensive.

Is the application deployed across multiple Azure subscriptions?

Understanding the subscription landscape of the application and how components are organized within or across subscriptions is important when analyzing if relevant subscription limits or quotas can be navigated. Review Azure subscription and service limits to validate that required capacity is within Azure service scale limits and quotas. To learn more, see Azure subscription and service limits.

Next step

Resiliency and dependencies

Related links

- For information on minimizing dependencies, see Minimize coordination.
- For more information on fault-points and fault-modes, see Failure Mode Analysis for Azure applications.
- For information on managed services, see Use platform as a service (PaaS) options.

Go back to the main article: Design

Resiliency and dependencies

11/19/2021 • 4 minutes to read • Edit Online

Building failure recovery into the system should be part of the architecture and design phases from the beginning to avoid the risk of failure. Dependencies are required for the application to fully operate.

Key points

- Identify possible failure points in the system with failure mode analysis.
- Eliminate all single point of failure.
- Maintain a complete list of application dependencies.
- Ensure that applications can operate in the absence of their dependencies.
- Understand the SLA of individual components within the system to define reliability targets.

Build resiliency with failure mode analysis

Failure mode analysis (FMA) is a process for building resiliency into a system, by identifying possible failure points in the system. The FMA should be part of the architecture and design phases, so that you can build failure recovery into the system from the beginning.

Identify all fault-points and fault-modes. Fault-points describe the elements within an application architecture which are capable of failing, while fault-modes capture the various ways by which a fault-point may fail. To ensure an application is resilient to end-to-end failures, it is essential that all fault-points and fault-modes are understood and operationalized. To learn more, see Failure mode analysis for Azure applications.

Eliminate all single point of failure. A single point of failure describes a specific fault-point which if it where to fail, would bring down the entire application. Single points of failure introduce significant risk since any failure of this component will cause an application outage. To learn more, see Make all things redundant.

> **NOTE**
>
> Eliminate all *singletons*. A singleton describes a logical component within an application for which there can only be a single instance. It can apply to stateful architectural components or application code constructs. Ultimately, singletons introduce a significant risk by creating single points of failure within the application design.

Understand the impact of dependencies

Internal dependencies describe components within the application scope which are required for the application to fully operate. *External* dependencies capture required components outside the scope of the application, such as another application or third-party service. Dependencies may be categorized as either strong or weak based on whether or not the application is able to continue operating in a degraded fashion in their absence. To learn more, see Twelve-Factor App: Dependencies.

You should maintain a complete list of application dependencies. Examples of typical dependencies include platform dependencies outside the remit of the application, such as Azure Active Directory, Express Route, or a central NVA (Network Virtual Appliance), as well as application dependencies such as APIs. For cost purposes, it's important to understand the price for these services and how they are being charged. For more details see Cost models.

You can map application dependencies either as a simple list or a document. Usually this is part of a design

document or reference architecture.

Understand the impact of an outage with each dependency. Strong dependencies play a critical role in application function and availability. Their absence will have a significant impact, while the absence of weak dependencies may only impact specific features and not affect overall availability. This reflects the cost that is needed to maintain the High Availability relationship between the service and its dependencies. Classifying dependencies as either strong or weak will help you identify which components are essential to the application.

Maintain SLAs and support agreements for critical dependencies. A Service Level Agreement (SLA) represents a commitment around performance and availability of the application. Understanding the SLA of individual components within the system is essential in order to define reliability targets. Knowing the SLA of dependencies will also provide a justification for additional spend when making the dependencies highly available and with proper support contracts. The operational commitments of all external and internal dependencies should be understood to inform the broader application operations and health model.

The usage of platform level dependencies such as Azure Active Directory must also be understood to ensure that their availability and recovery targets align with that of the application

Ensure that applications can operate in the absence of their dependencies. If the application has strong dependencies which it cannot operate in the absence of, then the availability and recovery targets of these dependencies should align with that of the application itself. Make an effort to minimize dependencies to achieve control over application reliability. To learn more, see Minimize dependencies.

Ensure that the lifecycle of the application decoupled from its dependencies. If the application lifecycle is closely coupled with that of its dependencies, it can limit the operational agility of the application. This is true particularly where new releases are concerned.

Next step

Best practices

Related links

- For information on failure mode analysis, see Failure mode analysis for Azure applications.
- For information on single point of failure, see Make all things redundant.
- For information on fault-points and fault-modes, see Failure Mode Analysis for Azure applications.
- For information on minimizing dependencies, see Minimize coordination.

Go back to the main article: Design

Best practices for designing reliability in Azure applications

11/19/2021 • 2 minutes to read • Edit Online

This article lists Azure best practices to enhance designing Azure applications for reliability. These best practices are derived from our experience with Azure reliability and the experiences of customers like yourself.

During the architectural phase, focus on implementing practices that meet your business requirements, identify failure points, and minimize the scope of failures.

Build availability targets and recovery targets into your design

A Service Level Agreement (SLA), is an availability target that represents a commitment around performance and availability of the application. Understanding the SLA of individual components within the system is essential in order to define reliability targets. Recovery targets identify how long the workload can be unavailable and how much data is acceptable to lose during a disaster. Define target reports for the application and key scenarios. There may be penalties, such as financial charges, associated with failing to meet SLA commitments. The consequences of not satisfying availability targets should be fully understood.

Ensure the application & data platforms meet your reliability requirements

Designing application platform and data platform resiliency and availability are critical to ensuring overall application reliability.

Ensure connectivity

To ensure connection availability and improve reliability with Azure services:

- Use a global load balancer used to distribute traffic and/or failover across regions.
- For cross-premises connectivity (ExpressRoute or VPN) ensure there redundant connections from different locations.
- Simulate a failure path to ensure connectivity is available over alternative paths.
- Eliminate all single points of failure from the data path (on-premises and Azure.

Use zone-aware services

Zone-aware services can improve reliability and ensure availability during failure scenarios affecting a datacenter within a region. They can also be used to deploy gateway instances across zones for improved reliability and availability during failure scenarios affecting a datacenter within a region.

Design resilience to respond to outages

Applications should be designed to operate even when affected by regional, zonal, service or component failures across critical application scenarios and functionality. Application operations may experience reduced functionality or degraded performance during an outage.

Perform a failure mode analysis (FMA)

FMA builds resiliency into an application early in the design stage. It helps you identify the types of failures your application might experience, the potential effects of each, and possible recovery strategies.

Have all single points of failure been eliminated? A single point of failure describes a specific fault-point which if it where to fail would bring down the entire application. Single points of failure introduce significant risk since any failure of this component will cause an application outage.

Have all fault-points and fault-modes been identified? Fault-points describe the elements within an application architecture which are capable of failing, while fault-modes capture the various ways by which a fault-point may fail. To ensure an application is resilient to end-to-end failures, it is essential that all fault-points and fault-modes are understood and operationalized.

Understand the impact of an outage with each dependency

Strong dependencies play a critical role in application function and availability. Their absence will have a significant impact, while the absence of weak dependencies may only impact specific features and not affect overall availability. Dependencies may be categorized as either strong or weak based on whether or not the application is able to continue operating in a degraded fashion in their absence.

Design for scalability

A cloud application must be able to scale to accommodate changes in usage. Begin with discrete components, and design the application to respond automatically to load changes whenever possible. Keep scaling limits in mind during design so you can expand easily in the future.

Next step

Testing

> Go back to the main article: Design

Testing for reliability

11/19/2021 • 2 minutes to read • Edit Online

Regular testing should be performed as part of each major change and if possible, on a regular basis to validate existing thresholds, targets and assumptions. Testing should also ensure the validity of the health model, capacity model, and operational procedures.

Checklist

Have you tested your applications with reliability in mind?

- Test regularly to validate existing thresholds, targets and assumptions.
- Automate testing as much as possible.
- Perform testing on both key test environments with the production environment.
- Perform chaos testing by injecting faults.
- Create and test a disaster recovery plan on a regular basis using key failure scenarios.
- Design disaster recovery strategy to run most applications with reduced functionality.
- Design a backup strategy that is tailored to business requirements and circumstances of the application.
- Test and validate the failover and failback approach successfully at least once.
- Configure request timeouts to manage inter-component calls.
- Implement retry logic to handle transient application failures and transient failures with internal or external dependencies.
- Configure and test health probes for your load balancers and traffic managers.
- Apply chaos principles continuously.
- Create and organize a central chaos engineering team.

Azure services

- Azure Site Recovery
- Azure Pipelines
- Azure Traffic Manager
- Azure Load Balancer

Reference architecture

- Failure Mode Analysis for Azure applications
- High availability and disaster recovery scenarios for IaaS apps
- Back up files and applications on Azure Stack Hub

Next step

Resiliency testing

Related links

- For information on performance testing, see Performance testing.
- For information on chaos engineering, see Chaos engineering.

- For information on failure and disaster recovery, see Failure and disaster recovery for Azure applications.
- For information on testing applications, see Testing your application and Azure environment.

Testing applications for availability and resiliency

11/19/2021 • 3 minutes to read • Edit Online

Applications should be tested to ensure *availability* and *resiliency*. Availability describes the amount of time when an application runs in a healthy state without significant downtime. Resiliency describes how quickly an application recovers from failure.

Being able to measure availability and resiliency can answer questions like, How much downtime is acceptable? How much does potential downtime cost your business? What are your availability requirements? How much do you invest in making your application highly available? What is the risk versus the cost? Testing plays a critical role in making sure your applications can meet these requirements.

Key points

- Test regularly to validate existing thresholds, targets and assumptions.
- Automate testing as much as possible.
- Perform testing on both key test environments with the production environment.
- Verify how the end-to-end workload performs under intermittent failure conditions.
- Test the application against critical non-functional requirements for performance.
- Conduct load testing with expected peak volumes to test scalability and performance under load.
- Perform chaos testing by injecting faults.

When to test

Regular testing should be performed as part of each major change and if possible, on a regular basis to validate existing thresholds, targets and assumptions. While the majority of testing should be performed within the testing and staging environments, it is often beneficial to also run a subset of tests against the production system. Plan a 1:1 parity of key test environments with the production environment.

> **NOTE**
>
> Automate testing where possible to ensure consistent test coverage and reproducibility. Automate common testing tasks and integrate them into your build processes. Manually testing software is tedious and susceptible to error, although manual explorative testing may also be conducted.

Testing for resiliency

To test resiliency, you should verify how the end-to-end workload performs under intermittent failure conditions.

Run tests in production using both synthetic and real user data. Test and production are rarely identical, so it's important to validate your application in production using a blue-green or canary deployment. This way, you're testing the application under real conditions, so you can be sure that it will function as expected when fully deployed.

As part of your test plan, include:

- Chaos engineering
- Automated pre-deployment testing

- Fault injection testing
- Peak load testing
- Disaster recovery testing

Performance testing

The primary goal of performance testing is to validate benchmark behavior for the application. Performance testing is the superset of both *load testing* and *stress testing*.

Load testing validates application scalability by rapidly and/or gradually increasing the load on the application until it reaches a threshold/limit. Stress testing involves various activities to overload existing resources and remove components to understand overall resiliency and how the application responds to issues.

Simulation testing

Simulation testing involves creating small, real-life situations. Simulations demonstrate the effectiveness of the solutions in the recovery plan and highlight any issues that weren't adequately addressed.

As you perform simulation testing, follow best practices:

- Conduct simulations in a manner that doesn't disrupt actual business but feels like a real situation.
- Make sure that simulated scenarios are completely controllable. If the recovery plan seems to be failing, you can restore the situation back to normal without causing damage.
- Inform management about when and how the simulation exercises will be conducted. Your plan should detail the time frame and the resources affected during the simulation.

Fault injection testing

For fault injection testing, check the resiliency of the system during failures, either by triggering actual failures or by simulating them. Here are some strategies to induce failures:

- Shut down virtual machine (VM) instances.
- Crash processes.
- Expire certificates.
- Change access keys.
- Shut down the DNS service on domain controllers.
- Limit available system resources, such as RAM or number of threads.
- Unmount disks.
- Redeploy a VM.

Your test plan should incorporate possible failure points identified during the design phase, in addition to common failure scenarios:

- Test your application in an environment as close to production as possible.
- Test failures in combination.
- Measure the recovery times, and be sure that your business requirements are met.
- Verify that failures don't cascade and are handled in an isolated way.

Test under peak loads

Load testing is crucial for identifying failures that only happen under load, such as the back-end database being overwhelmed or service throttling. Test for peak load and anticipated increase in peak load, using production data or synthetic data that is as close to production data as possible. Your goal is to see how the application behaves under real-world conditions.

Next step

Backup and recovery

Related links

- For more test types, see Test types.
- To learn about load and stress tests, see Performance testing.
- To learn about chaos testing, see Chaos engineering.

Go back to the main article: Testing

Backup and disaster recovery for Azure applications

11/19/2021 • 6 minutes to read • Edit Online

Disaster recovery is the process of restoring application functionality in the wake of a catastrophic loss.

In the cloud, we acknowledge up front that failures will happen. Instead of trying to prevent failures altogether, the goal is to minimize the effects of a single failing component. Testing is one way to minimize these effects. Automate testing your applications where possible, but you need to be prepared for when they fail. When a failure happens, having backup and recovery strategies becomes important.

Your tolerance for reduced functionality during a disaster is a business decision that varies from one application to the next. It might be acceptable for some applications to be unavailable or to be partially available with reduced functionality or delayed processing for a while. For other applications, any reduced functionality is unacceptable.

Key points

- Create and test a disaster recovery plan regularly using key failure scenarios.
- Design disaster recovery strategy to run most applications with reduced functionality.
- Design a backup strategy that is tailored to business requirements and circumstances of the application.
- Automate failover and failback steps and processes.
- Test and validate the failover and failback approach successfully at least once.

Disaster recovery plan

Start by creating a recovery plan. The plan is considered complete after it has been fully tested. Include the people, processes, and applications needed to restore functionality within the service-level agreement (SLA) you've defined for your customers.

Consider the following suggestions when creating and testing your disaster recovery plan:

- Include the process for contacting support and for escalating issues. This information will help to avoid prolonged downtime as you work out the recovery process for the first time.
- Evaluate the business impact of application failures.
- Choose a cross-region recovery architecture for mission-critical applications.
- Identify a specific owner of the disaster recovery plan, including automation and testing.
- Document the process, especially any manual steps.
- Automate the process as much as possible.
- Establish a backup strategy for all reference and transactional data, and test backup restoration regularly.
- Set up alerts for the stack of the Azure services consumed by your application.
- Train operations staff to execute the plan.
- Perform regular disaster simulations to validate and improve the plan.

If you're using Azure Site Recovery to replicate virtual machines (VMs), create a fully automated recovery plan to fail over the entire application.

Operational readiness testing

Perform an operational readiness test for failover to the secondary region and for failback to the primary region. Many Azure services support manual failover or test failover for disaster recovery drills. Instead, you can

simulate an outage by shutting down or removing Azure services.

Automated operational responses should be tested frequently as part of the normal application lifecycle to ensure operational effectiveness.

Failover and failback testing

Test failover and failback to verify that your application's dependent services come back up in a synchronized manner during disaster recovery. Changes to systems and operations may affect failover and failback functions, but the impact may not be detected until the main system fails or becomes overloaded. Test failover capabilities *before* they're required to compensate for a live problem. Also, be sure dependent services failover and failback in the correct order.

If you're using Azure Site Recovery to replicate VMs, run disaster recovery drills periodically by testing failovers to validate your replication strategy. A test failover doesn't affect the ongoing VM replication or your production environment. For more information, see Run a disaster recovery drill to Azure.

Dependent service outage

For each dependent service, you should understand the implications of service disruption and the way that the application will respond. Many services include features that support resiliency and availability, so evaluating each service independently is likely to improve your disaster recovery plan. For example, Azure Event Hubs supports failing over to the secondary namespace.

Network outage

When parts of the Azure network are inaccessible, you may not be able to access your application or data. In this situation, we recommend designing the disaster recovery strategy to run most applications with reduced functionality.

If reducing functionality isn't an option, the remaining options are application downtime or failover to an alternate region.

In a reduced functionality scenario:

- If your application can't access its data because of an Azure network outage, you can run locally with reduced application functionality by using cached data.
- You can store data in an alternate location until connectivity is restored.

Recovery automation

The steps required to recover or failover the application to a secondary Azure region in failure situations should be codified, preferably in an automated manner, to ensure capabilities exist to respond effectively to an outage in a way that limits impact. Similar codified steps should also exist to capture the process required to failback the application to the primary region once a failover triggering issue has been addressed.

When automating failover procedures, ensure that the tooling used for orchestrating the failover is also considered in the failover strategy. For example, if you run your failover from Jenkins running on a VM, you'll be in trouble if that virtual machine is part of the outage. Azure DevOps Projects are scoped to a region too.

Backup strategy

Many alternative strategies are available for implementing distributed compute across regions. These strategies must be tailored to the specific business requirements and circumstances of the application. At a high level, the approaches can be divided into the following categories:

- **Redeploy on disaster**: In this approach, the application is redeployed from scratch at the time of disaster. Redeploying from scratch is appropriate for non-critical applications that don't require a guaranteed recovery time.
- **Warm Spare (Active/Passive)**: Create a secondary hosted service in an alternate region, and deploy roles to guarantee minimal capacity. However, the roles don't receive production traffic. This approach is useful for applications that have not been designed to distribute traffic across regions.
- **Hot Spare (Active/Active)**: The application is designed to receive production load in multiple regions. The cloud services in each region might be configured for higher capacity than required for disaster recovery purposes. Instead, the cloud services might scale out as necessary at the time of a disaster and failover. This approach requires a large investment in application design, but it has significant benefits. These include low and guaranteed recovery time, continuous testing of all recovery locations, and efficient usage of capacity.

Plan for regional failures

Azure is divided physically and logically into units called regions. A region consists of one or more data centers in close proximity. Many regions and services also support availability zones, which can be used to provide more resiliency against outages in a single data center. Consider using regions with availability zones to improve the availability of your solution.

Under rare circumstances, it's possible that facilities in an entire availability zone or region can become inaccessible, for example, because of network failures. Or, facilities can be lost entirely, for example, because of a natural disaster. Azure has capabilities for creating applications that are distributed across zones and regions. Such distribution helps to minimize the possibility that a failure in one zone or region could affect other zones or regions.

Next step

Automatic retry of failed backup jobs

Related links

- For information on testing failovers, see Run a disaster recovery drill to Azure.
- For information on Event Hubs, see Azure Event Hubs.

Go back to the main article: Testing

Automatic retry of failed backup jobs

11/19/2021 • 5 minutes to read • Edit Online

Azure Backup comprehensively protects your data assets in Azure through a simple, secure, and cost-effective solution that requires zero infrastructure. Azure's built-in data protection offers a solution for a wide range of workloads, and helps protect your mission critical workloads running in the cloud. Azure's built-in data ensures your backups are always available and managed at scale across your entire backup estate.

As a backup user or administrator, you can monitor all backup solutions and configure alerts to notify you about important events.

Many of the failures or outages are transient. You can solve them just by retrying the backup, or the restore job. However, waiting for an engineer to retry the job manually or assign the relevant permission wastes valuable time. Automation is the smarter way to retry the failed jobs, and ensures you continue to meet your target RPOs with one successful backup a day.

Retrieve relevant backup data through Azure Resource Graph (ARG) and combine the data with corrective PowerShell and CLI steps. This article guides you through retrying the backup for all failed jobs using ARG and PowerShell.

Prerequisites

You'll need an Azure Automation account. You can use an existing account or create a new account with one user-assigned managed identity, at minimum.

Assign permissions to managed identities

To assign permissions to managed identities to stop and start a virtual machine, complete the following steps:

1. Sign in to Azure interactively using the Connect-AzAccount `cmdlet` and follow the instructions:

```
# Sign in to your Azure subscription

$sub = Get-AzSubscription -ErrorAction SilentlyContinue
if(-not($sub))
{
 Connect-AzAccount
}

# If you have multiple subscriptions, set the one to use
# Select-AzSubscription -SubscriptionId <SUBSCRIPTIONID>
```

2. Provide an appropriate value for the following variables and then run the script:

```
$resourceGroup = "resourceGroupName"

# These values are used in this tutorial
$automationAccount = "xAutomationAccount"
$userAssignedManagedIdentity = "xUAMI"
```

3. Use the PowerShell `cmdlet` New-AzRoleAssignment to assign a role to the system-assigned managed identity:

```
$role1 = "DevTest Labs User"

$SAMI = (Get-AzAutomationAccount -ResourceGroupName $resourceGroup
-Name $automationAccount).Identity.PrincipalId
New-AzRoleAssignment
 -ObjectId $SAMI
 -ResourceGroupName $resourceGroup
 -RoleDefinitionName $role1
```

4. You need the same role assignment for the user-assigned managed identity:

```
$UAMI = (Get-AzUserAssignedIdentity -ResourceGroupName
$resourceGroup -Name $userAssignedManagedIdentity).PrincipalId
New-AzRoleAssignment
    -ObjectId $UAMI
    -ResourceGroupName $resourceGroup
    -RoleDefinitionName $role1
```

5. You'll need extra permissions for the system-assigned managed identity to run the `cmdlets`: `Get-AzUserAssignedIdentity` and `Get-AzAutomationAccount`:

```
$role2 = "Reader"
New-AzRoleAssignment
 -ObjectId $SAMI
 -ResourceGroupName $resourceGroup
 -RoleDefinitionName $role2
```

Add modules

For the preceding scripts to work, install the following modules by navigating to the individual module gallery after you've created the automation account:

- Az.Accounts
- Az.RecoveryServices
- Az.Graph

Create a PowerShell runbook

The runbook starts a stopped VM, or stops a running VM. To create a runbook that managed identities can run, complete the following steps:

1. Sign in to the Azure portal and navigate to your Automation account.
2. Under **Process Automation**, select **Runbooks**.
3. Select **Create a runbook**:
   a. Name the runbook *miTesting*.
   b. From the **Runbook type** dropdown menu, select **PowerShell**.
   c. Select **Create**.
4. In the runbook editor, paste the following code:

```
$connection = Get-AutomationConnection -Name AzureRunAsConnection

$connectionResult = Connect-AzAccount
-ServicePrincipal
-Tenant $connection.TenantID
```

```
-Tenant $connection.TenantID
-ApplicationId $connection.ApplicationID
-CertificateThumbprint $connection.CertificateThumbprint
"Login successful.."

$query = "RecoveryServicesResources
| where type in~ ('microsoft.recoveryservices/vaults/backupjobs')
| extend vaultName = case(type =~
'microsoft.dataprotection/backupVaults/backupJobs',properties.vaultName,type =~
'Microsoft.RecoveryServices/vaults/backupJobs',split(split(id, '/Microsoft.RecoveryServices/vaults/')
[1],'/')[0],'--')
| extend friendlyName = case(type =~
'microsoft.dataprotection/backupVaults/backupJobs',strcat(properties.dataSourceSetName , '/',
properties.dataSourceName),type =~ 'Microsoft.RecoveryServices/vaults/backupJobs',
properties.entityFriendlyName, '--')
| extend dataSourceType = case(type =~
'Microsoft.RecoveryServices/vaults/backupJobs',properties.backupManagementType,type =~
'microsoft.dataprotection/backupVaults/backupJobs',properties.dataSourceType,'--')
| extend protectedItemName = split(split(properties.backupInstanceId, 'protectedItems')[1],'/')[1]
| extend vaultId = tostring(split(id, '/backupJobs')[0])
| extend vaultSub = tostring( split(id, '/')[2])
| extend jobStatus = case (properties.status == 'Completed' or properties.status ==
'CompletedWithWarnings','Succeeded',properties.status == 'Failed','Failed',properties.status ==
'InProgress', 'Started', properties.status), operation = case(type =~
'microsoft.dataprotection/backupVaults/backupJobs' and tolower(properties.operationCategory) =~
'backup' and properties.isUserTriggered == 'true',strcat('adhoc',properties.operationCategory),type
=~ 'microsoft.dataprotection/backupVaults/backupJobs', tolower(properties.operationCategory), type =~
'Microsoft.RecoveryServices/vaults/backupJobs' and tolower(properties.operation) =~ 'backup' and
properties.isUserTriggered == 'true',strcat('adhoc',properties.operation),type =~
'Microsoft.RecoveryServices/vaults/backupJobs',tolower(properties.operation), '--'),startTime =
todatetime(properties.startTime),endTime = properties.endTime, duration = properties.duration
| where startTime >= ago(24h)
| where (dataSourceType in~ ('AzureIaasVM'))
| where jobStatus=='Failed'
| where operation == 'backup' or operation == 'adhocBackup'
| project vaultSub, vaultId, protectedItemName, startTime, endTime, jobStatus, operation
| sort by vaultSub"

$subscriptions = Get-AzSubscription | foreach {$_.SubscriptionId}

$result = Search-AzGraph -Subscription $subscriptions -Query $query -First 5

$result = $result.data


$prevsub = ""
foreach($jobresponse in $result)
{

               if($jobresponse.vaultSub -ne $prevsub)

            {

                        Set-AzContext -SubscriptionId
$jobresponse.vaultSub

                        $prevsub = $jobresponse.vaultSub

            }

            $item = Get-AzRecoveryServicesBackupItem -VaultId
$jobresponse.vaultId -BackupManagementType AzureVM -WorkloadType AzureVM -Name
$jobresponse.protectedItemName



            Backup-AzRecoveryServicesBackupItem -ExpiryDateTimeUTC
(get-date).AddDays(10) -Item $item -VaultId $jobresponse.vaultId

}
```

```
}
```

5. Select **Save** and then **Test pane**.

You've now successfully created a PowerShell runbook.

Create a new schedule with PowerShell

To create a new schedule with PowerShell, you must:

- Use the New-AzAutomationSchedule `cmdlet` to create schedules.
- Specify the start time for the schedule and the frequency it should run.

The following code example shows how to create a recurring schedule that runs every day at `1:00 PM` for one year:

```
PS C:\> $StartTime = Get-Date "13:00:00"
PS C:\> $EndTime = $StartTime.AddYears(1)
PS C:\> New-AzAutomationSchedule -AutomationAccountName " MyAutomationAccount" -Name "Schedule02" -StartTime
$StartTime -ExpiryTime $EndTime -DayInterval 1 -ResourceGroupName "ResourceGroup01"
```

- The first command creates a date object using the `Get-Date` `cmdlet` and then stores the object in the `$StartDate` variable. Specify a time that is, at least, five minutes in the future.
- The second command creates a date object using the `Get-Date` `cmdlet` and then stores the object in the `$EndDate` variable. The command specifies a future time.
- The final command creates a daily schedule named *Schedule02* to begin at the time stored in `$StartDate` and expires at the time stored in `$EndDate`.

Link a schedule to a runbook

Consider the following concepts when you link a schedule to a runbook:

- You can link a runbook to multiple schedules and a schedule can have multiple runbooks linked to it.
- If a runbook has parameters, you can provide values for them.
- Provide values for any mandatory parameters and any optional parameters. These values are used each time the runbook is started by this schedule.
- You can attach the same runbook to another schedule and specify different parameter values.

Link a schedule to a runbook with PowerShell

To link a schedule to a runbook with PowerShell, you must:

- Use the Register-AzAutomationScheduledRunbook `cmdlet` to link a schedule.
- You can specify parameter values for the runbook with the Parameters $parameter.

For more information about how to specify parameter values, reference Starting a Runbook in Azure Automation.

The following code example shows how to link a schedule to a runbook by using an Azure Resource Manager `cmdlet` with parameters:

```
$automationAccountName = "MyAutomationAccount"
$runbookName = "Test-Runbook"
$scheduleName = "Sample-DailySchedule"
$params = @{"FirstName"="Joe";"LastName"="Smith";"RepeatCount"=2;"Show"=$true}
Register-AzAutomationScheduledRunbook -AutomationAccountName $automationAccountName
-Name $runbookName -ScheduleName $scheduleName -Parameters $params `
-ResourceGroupName "ResourceGroup01"
```

Next steps

Error handling

Error handling for resilient applications in Azure

11/19/2021 • 3 minutes to read • Edit Online

Ensuring your application can recover from errors is critical when working in a distributed system. You test your applications to prevent errors and failure, but you need to be prepared for when applications encounter issues or fail. Understanding how to handle errors and prevent potential failure becomes important, as testing doesn't always catch everything.

Many things in a distributed system are outside your span of control and your means to test. This can be the underlying cloud infrastructure, third party runtime dependencies, etc. You can be sure something will fail eventually, so you need to prepare for that.

Key points

- Uncover issues or failures in your application's retry logic.
- Configure request timeouts to manage inter-component calls.
- Implement retry logic to handle transient application failures and transient failures with internal or external dependencies.
- Configure and test health probes for your load balancers and traffic managers.
- Segregate read operations from update operations across application data stores.

Transient fault handling

Track the number of transient exceptions and retries over time to uncover issues or failures in your application's retry logic. A trend of increasing exceptions over time may indicate that the service is having an issue and may fail. To learn more, see Retry service specific guidance.

Use the Retry pattern, paying particular attention to issues and considerations. Avoid overwhelming dependent services by implementing the Circuit Breaker pattern. Review and incorporate additional best practices guidance for Transient fault handling. While calling systems that have Throttling pattern implemented, ensure that your retries are not counter productive.

A reference implementation is available here. It uses Polly and IHttpClientBuilder to implement the Circuit Breaker pattern.

Request timeouts

When making a service call or a database call, ensure that appropriate request timeouts are set. Database Connection timeouts are typically set to 30 seconds. For guidance on how to troubleshoot, diagnose, and prevent SQL connection errors, see transient errors for SQL Database.

Leverage design patterns that encapsulate robust timeout strategies like Choreography pattern or Compensating Transaction pattern.

A reference implementation is available on GitHub.

Cascading Failures

The Circuit Breaker pattern provides stability while the system recovers from a failure and minimizes the impact on performance. It can help to maintain the response time of the system by quickly rejecting a request for an

operation that's likely to fail, rather than waiting for the operation to time out, or never return.

A circuit breaker might be able to test the health of a service by sending a request to an endpoint exposed by the service. The service should return information indicating its status.

Retry pattern. Describes how an application can handle anticipated temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that has previously failed.

Samples related to this pattern are here.

Application Health Probes

Configure and test health probes for your load balancers and traffic managers. Ensure that your health endpoint checks the critical parts of the system and responds appropriately.

- For Azure Front Door and Azure Traffic Manager, the health probe determines whether to fail over to another region. Your health endpoint should check any critical dependencies that are deployed within the same region.
- For Azure Load Balancer, the health probe determines whether to remove a VM from rotation. The health endpoint should report the health of the VM. Don't include other tiers or external services. Otherwise, a failure that occurs outside the VM will cause the load balancer to remove the VM from rotation.

Samples related to heath probes are here.

- ARM template that deploys an Azure Load Balancer and health probes that detect the health of the sample service endpoint.
- An ASP.NET Core Web API that shows configuration of health checks at startup.

Command and Query Responsibility Segregation (CQRS)

Achieve levels of scale and performance needed for your solution by segregating read and write interfaces by implementing the CQRS pattern.

Next step

Chaos engineering

Related links

- For information on transient faults, see Troubleshoot transient connection errors.
- For guidance on implementing health monitoring in your application, see Health Endpoint Monitoring pattern.

Go back to the main article: Testing

Chaos engineering

11/19/2021 • 5 minutes to read • Edit Online

Chaos engineering is a methodology that helps developers attain consistent reliability by hardening services against failures in production. Another way to think about chaos engineering is that it's about embracing the inherent chaos in complex systems and, through experimentation, growing confidence in your solution's ability to handle it.

A common way to introduce chaos is to deliberately inject faults that cause system components to fail. The goal is to observe, monitor, respond to, and improve your system's reliability under adverse circumstances. For example, taking dependencies offline (stopping API apps, shutting down VMs, etc.), restricting access (enabling firewall rules, changing connection strings, etc.), or forcing failover (database level, Front Door, etc.), is a good way to validate that the application is able to handle faults gracefully.

It's difficult to simulate the characteristics of a service's behavior at scale outside a production environment. The transient nature of cloud platforms can exacerbate this difficulty. Architecting your service to expect failure is a core approach to creating a modern service. Chaos engineering embraces the uncertainty of the production environment and strives to anticipate rare, unpredictable, and disruptive outcomes, so that you can minimize any potential impact on your customers.

Key points

- Increase service resiliency and ability to react to failures.
- Apply chaos principles continuously.
- Create and organize a central chaos engineering team.
- Follow best practices for chaos testing.

Increase resiliency

Chaos engineering is aimed at increasing your service's resiliency and its ability to react to failures. By conducting experiments in a controlled environment, you can identify issues that are likely to arise during development and deployment. During this process, be vigilant in adopting the following guidelines:

- Be proactive.
- Embrace failure.
- Break the system.
- Identify and address single points of failure early.
- Install guardrails and graceful mitigation.
- Minimize the blast radius.
- Build immunity.

Chaos engineering should be an integral part of development team culture and an ongoing practice, not a short-term tactical effort in response to a single outage.

Development team members are partners in the process. They must be equipped with the resources to triage issues, implement the testability that's required for fault injection, and drive the necessary product changes.

When to apply chaos

Ideally, you should apply chaos principles continuously. There's constant change in the environments in which

software and hardware run, so monitoring the changes is key. By constantly applying stress or faults on components, you can help expose issues early, before small problems are compounded by a number of other factors.

Apply chaos engineering principles when you're:

- Deploying new code.
- Adding dependencies.
- Observing changes in usage patterns.
- Mitigating problems.

Process

Chaos engineering requires specialized expertise, technology, and practices. As with security and performance teams, the model of a central team supporting the service teams is a common, effective approach.

If you plan to practice the simulated handling of potentially catastrophic scenarios under controlled conditions, here's a simplified way to organize your teams:

ATTACKER	DEFENDER FOR CLOUD
Inject faults	Assess
Provide hints	Analyze
	Mitigate

Goals

- Familiarize team members with monitoring tools.
- Recognize outage patterns.
- Learn how to assess the impact.
- Determine the root cause and mitigate accordingly.
- Practice log analysis.

Overall method

1. Start with a hypothesis.
2. Measure baseline behavior.
3. Inject a fault or faults.
4. Monitor the resulting behavior.
5. Document the process and observations.
6. Identify and act on the result.

Periodically validate your process, architecture choices, and code. By conducting fault-injection experiments, you can confirm that monitoring is in place and alerts are set up, the *directly responsible individual* (DRI) process is effective, and your documentation and investigation processes are up to date. Keep in mind a few key considerations:

- Challenge system assumptions.
- Validate change (topology, platform, resources).
- Use service-level agreement (SLA) buffers.
- Use live-site outages as opportunities.

Best practices

Shift left

Shift-left testing means experiment early, experiment often. Incorporate fault-injection configurations and create resiliency-validation gates during the development stages and in the deployment pipeline.

Shift right

Shift-right testing means that you verify that the service is resilient where it counts in a pre-production or production environment with actual customer load. Adopt a proactive approach as opposed to reacting to failures. Be a part of determining and controlling requirements for the blast radius.

Blast radius

Stop the experiment when it goes beyond scope. Unknown results are an expected outcome of chaos experiments. Strive to achieve balance between collecting substantial result data and affecting as few production users as possible. For an example of this principle in practice, see the Bulkhead pattern article.

Error budget testing

Establish an error budget as an investment in chaos and fault injection. Your error budget is the difference between achieving 100% of the service-level objective (SLO) and achieving the *agreed-upon* SLO.

Considerations for chaos testing

The following questions and answers discuss considerations about chaos engineering, based on its application inside Azure.

Have you identified faults that are relevant to the development team?

Work closely with the development teams to ensure the relevance of the injected failures. Use past incidents or issues as a guide. Examine dependencies and evaluate the results when those dependencies are removed.

An external team can't hypothesize faults for your team. A study of failures from an artificial source might be relevant to your team's purposes, but the effort must be justified.

Have you injected faults in a way that accurately reflects production failures?

Simulate production failures. Treat injected faults in the same way that you would treat production-level faults. Enforcing a tighter limit on the blast radius will enable you to simulate a production environment. Each fault-injection effort must be accompanied by tooling that's designed to inject the types of faults that are relevant to your team's scenarios. Here are two basic ways:

- Inject faults in a non-production environment, such as Canary or Test In Production (TIP).
- Partition the production service or environment.

Halt all faults and roll back the state to its last-known good configuration if the state seems severe.

Have you built confidence incrementally?

Start by hardening the core, and then expand out in layers. At each point, lock in progress with automated regression tests. Each team should have a long-term strategy based on a progression that makes sense for the team's circumstances.

By applying the shift left strategy, you can help ensure that any obstacles to developer usage are removed early and the testing results are actionable.

The process must be very *low tax*. That is, the process must make it easy for developers to understand what happened and to fix the issues. The effort must fit easily into their normal workflow, not burden them with one-

off special activities.

Next step

Best practices

Related links

- For information on release testing, see Testing your application and Azure environment.
- For more information, see Bulkhead pattern.

Go back to the main article: Testing

Testing best practices for reliability in Azure applications

11/19/2021 • 2 minutes to read • Edit Online

This article lists Azure best practices to enhance testing Azure applications for reliability. These best practices are derived from our experience with Azure reliability and the experiences of customers like yourself.

During the architectural phase, focus on implementing practices that meet your business requirements, and ensure that applications will run in a healthy state without significant downtime.

Test regularly

Test regularly to validate existing thresholds, targets, and assumptions. Regular testing should be performed as part of each major change and if possible, on a regular basis. While most testing should be performed within the testing and staging environments, it is often beneficial to also run a subset of tests against the production system.

Test for resiliency

To test resiliency, you should verify how the end-to-end workload performs under intermittent failure conditions. Consider performing the following tests:

- Performance testing
- Simulation testing
- Fault injection testing
- Load testing
- Operational readiness testing
- Failover and failback testing

Design a backup strategy

Design a backup strategy that is tailored to the specific business requirements and circumstances of the application. At a high level, the approaches can be divided into these categories: 1) Redeploy on disaster, 2) Warm Spare (Active/Passive), and 3) Hot Spare (Active/Active).

Design a disaster recovery strategy

When parts of the Azure network are inaccessible, you might not be able to access your application or data. In this situation, design a disaster recovery strategy to run most applications with reduced functionality.

Codify steps to failover and fallback

Codify steps, preferably automatically, to failover and fallback the application to the primary region once a failover triggering issue has been addressed. Doing this should ensure capabilities exist to effectively respond to an outage in a way that limits impact.

Plan for regional failures

Use Azure for creating applications that are distributed across regions. Such distribution helps to minimize the

possibility that a failure in one region could affect other regions.

Implement retry logic

Track the number of transient exceptions and retries over time to uncover issues or failures in your application's retry logic. A trend of increasing exceptions over time may indicate that the service is having an issue and may fail.

Configure and test health probes

Configure and test health probes for your load balancers and traffic managers. Ensure that your health endpoint checks the critical parts of the system and responds appropriately.

Segregate read and write interfaces

Achieve levels of scale and performance needed for your solution by segregating read and write interfaces by implementing the Command and Query Responsibility Segregation (CQRS) pattern.

Next step

Monitoring

> Go back to the main article: Testing

Monitoring for reliability

11/19/2021 • 2 minutes to read • Edit Online

Monitoring and diagnostics are crucial for resiliency. If something fails, you need to know *that* it failed, *when* it failed — and *why*.

Checklist

How do you monitor and measure application health?

- The application is instrumented with semantic logs and metrics.
- Application logs are correlated across components.
- All components are monitored and correlated with application telemetry.
- Key metrics, thresholds, and indicators are defined and captured.
- A health model has been defined based on performance, availability, and recovery targets.
- Azure Service Health events are used to alert on applicable service level events.
- Azure Resource Health events are used to alert on resource health events.
- Monitor long-running workflows for failures.

Azure services for monitoring

- Azure Monitor
- Application Insights
- Azure Service Health
- Azure Resource Health
- Azure Resource Manager
- Azure Policy

Reference architecture

- Hybrid availability and performance monitoring
- Unified logging for microservices applications

Next step

Application health

Related links

- Azure Monitor
- Continuous monitoring

Monitoring application health for reliability

11/19/2021 • 6 minutes to read • Edit Online

Monitoring and diagnostics are crucial for availability and resiliency. If something fails, you need to know *that* it failed, *when* it failed, and *why*.

Monitoring isn't the same as *failure detection*. For example, your application might detect a transient error and retry, avoiding downtime. But it should also log the retry operation so that you can monitor the error rate to get an overall picture of application health.

Key points

- Define alerts that are actionable and effectively prioritized.
- Create alerts that poll for services nearing their limits and quotas.
- Use application instrumentation to detect and resolve performance anomalies.
- Track the progress of long-running processes.
- Troubleshoot issues to gain an overall view of application health.

Alerting

Alerts are notifications of system health issues that are found during monitoring. Alerts only deliver value if they are actionable and effectively prioritized by on-call engineers through defined operational procedures. Present telemetry data in a dashboard or email alert format that makes it easy for an operator to notice problems or trends quickly.

Service level alerts

Use Azure Service Health to respond to *service level* events. Azure Service Health provides a view into the health of Azure services and regions. It issues communications that impact the following services:

- Outages
- Planned maintenance activities
- Other health advisories

Azure Service Health alerts should be configured to operationalize Service Health events. However, Service Health alerts shouldn't be used to detect issues because of associated latencies. There is a `5` minute service level objective (SLO) for automated issues, but many issues require manual interpretation to define a root cause analysis (RCA). Instead, alerts should be used to provide useful information to help interpret issues that have been detected and surfaced through the health model, to inform an operational response.

To learn more, reference Azure Service Health.

Resource level alerts

Use Azure Resource Health to respond to *resource level* events. Azure Resource Health provides information about the health of individual resources such as a specific virtual machine, and is highly useful when diagnosing unavailable resources.

Azure Resource Health alerts should be configured for specific resource groups and resource types. These alerts should be adjusted to maximize signal to noise ratios. For example, only distribute a notification when a resource becomes unhealthy according to the application health model or due to an Azure platform initiated event. It's important to consider transient issues when setting an appropriate threshold for resource unavailability. For example, configure an alert for a virtual machine with a threshold of `1` minute for

unavailability before an alert is triggered.

To learn more, reference Azure Resource Health.

Dashboards

You can also get a full-stack view of application state by using Azure dashboards to create a combined view of monitoring graphs from the following:

- Application Insights
- Log Analytics
- Azure Monitor metrics
- Service Health

Samples

Here are some samples about creating and querying alerts:

- HealthAlert: A sample about creating resource-level health activity log alerts. The sample uses Azure Resource Manager to create alerts.
- GraphAlertsPsSample: A set of PowerShell commands that queries for alerts generated against your subscription.

Azure subscription and service limits

Azure subscriptions have limits on certain resource types, such as number of resource groups, cores, and storage accounts. To ensure your application doesn't run up against Azure subscription limits, create alerts that poll for services nearing their limits and quotas.

Address the following subscription limits with alerts.

Individual services

Individual Azure services have consumption limits on:

- Storage
- Throughput
- Number of connections
- Requests per second

Your application will fail if it attempts to use resources beyond these limits, resulting in service throttling and possible downtime.

Depending on the specific service and your application requirements, you can often stay under these limits by scaling up (choosing another pricing tier, for example) or scaling out (adding new instances).

Azure storage scalability and performance targets

Azure allows a maximum number of storage accounts per subscription. If your application requires more storage accounts than are currently available in your subscription, create a new subscription with extra storage accounts. For more information, reference Azure subscription and service limits, quotas, and constraints.

Scalability targets for virtual machine disks

An Azure infrastructure as a service (IaaS) virtual machine supports attaching many data disks, depending on several factors, including the virtual machine size and the type of storage account. If your application exceeds the scalability targets for virtual machine disks, provision additional storage accounts and create the virtual machine disks there. To learn more, reference Scalability and performance targets for VM disks.

Virtual machine size

If the actual CPU, memory, disk, and I/O of your virtual machines approach the limits of the virtual machine size, your application may experience capacity issues. To correct the issues, increase the virtual machine size.

If your workload fluctuates over time, consider using virtual machine scale sets to automatically scale the number of virtual instances. Otherwise, you need to manually increase or decrease the number of virtual machines.

Azure SQL Database

If your Azure SQL Database tier isn't adequate to handle your application's Database Transaction Unit (DTU) requirements, your data use will be throttled. For more information on selecting the correct service plan, reference Azure SQL Database purchasing models.

Instrumentation

Instrument applications to measure the customer experience. Effective instrumentation is vital for detecting and resolving performance anomalies that can impact customer experience, and application availability. To build a robust application health model, it's vital that you achieve visibility into the operational state of critical internal dependencies, such as a shared NVA or Express Route connection.

Automated failover and failback systems depend on the correct functioning of monitoring and instrumentation. Dashboards that visualize system health and operator alerts also depend on having accurate monitoring and instrumentation. If these elements fail, miss critical information, or report inaccurate data, an operator might not realize that the system is unhealthy or failing. Make sure you include monitoring systems in your test plan.

Instrument applications to track calls to dependent services. Dependency tracking and measuring the duration or status of dependency calls is also vital to measuring overall application health. It should be used to inform a health model for the application.

Microsoft recommends collecting and storing logs, and key metrics of critical components.

Provide rich instrumentation:

- For failures that are likely, but have not yet occurred: provide enough data to determine the cause, mitigate the situation, and ensure that the system remains available.
- For failures that have already occurred: the application should return an appropriate error message to the user, but should attempt to continue running despite reduced functionality.

Monitoring systems should capture comprehensive details so that applications can be restored efficiently and, if necessary, designers and developers can modify the system to prevent the situation from recurring.

Long-running workflow failures

Long-running workflows often include multiple steps, each of which should be independent.

Track the progress of long-running processes to minimize the likelihood that the entire workflow will need to be rolled back or that multiple compensating transactions will need to be executed.

> **TIP**
>
> Monitor and manage the progress of long-running workflows by implementing a pattern such as Scheduler Agent Supervisor.

Analysis and diagnosis

Analyze data combined in these data stores to troubleshoot issues and gain an overall view of application health. Generally, you can search for and analyze the data in Application Insights, and Log Analytics using Kusto

queries, or view preconfigured graphs using management solutions. Use Azure Advisor to view recommendations with a focus on resiliency and performance.

Related links

- For information on dashboards, reference Azure dashboards.
- For information on virtual machine sizes, reference Sizes for virtual machines in Azure.
- For information on scale sets, reference virtual machine scale sets overview.

Go back to the main article: Monitoring

Next step

Health modeling

Health modeling for reliability

11/19/2021 • 5 minutes to read • Edit Online

The health model should be able to surface the health of critical system flows or key subsystems to ensure appropriate operational prioritization is applied. For example, the health model should be able to represent the current state of the user login transaction flow.

The health model should not treat all failures the same. For example, the health model should distinguish between transient and non-transient faults. It should clearly distinguish between expected-transient but recoverable failures and a true disaster state.

> **NOTE**
> The health model should clearly distinguish between expected-transient but recoverable failures and a true disaster state.

Key points

- Know how to tell if an application is healthy or unhealthy.
- Understand the impact of logs in diagnostic data.
- Ensure the consistent use of diagnostic settings across the application.
- Use critical system flows in your health model.

Healthy and unhealthy states

A health model qualifies what *healthy* and *unhealthy* states represent for the application. A holistic application health model should be used to quantify what healthy and unhealthy states represent across all application components. It's highly recommended that a "traffic light" model be used to indicate a green/healthy state when key non-functional requirements and targets are fully satisfied and resources are optimally utilized. For example, 95 percent of requests are processed in <= 500ms with AKS node utilization at x% etc. Once established, this health model should inform critical monitoring metrics across system components and operational sub-system composition.

The overall health state can be impacted by both application level issues and resource level failures. Telemetry correlation should be used to ensure transactions can be mapped through the end-to-end application and critical system flows, as this is vital to root cause analysis for failures. Platform level metrics and logs such as CPU percentage, network in/out, and disk operations/sec should be collected from the application to inform a health model and detect/predict issues. This can also help to distinguish between transient and non-transient faults.

Quantify application states

Application level events should be automatically correlated with resource level metrics to quantify the current application state. The overall health state can be impacted by both application level issues and resource level failures.

Telemetry correlation should be used to ensure transactions can be mapped through the end-to-end application and critical system flows, as this is vital to root cause analysis for failures. Platform level metrics and logs such as CPU percentage, network in/out, and disk operations/sec should be collected from the application to inform a health model and detect/predict issues(Telemetry correlation). This can also help to distinguish between

transient and non-transient faults.

Application logs

Application logs are an important source of diagnostics data. To gain insight when you need it most, follow these best practices for application logging:

- **Use semantic (structured) logging.** With structured logs, it's easier to automate the consumption and analysis of the log data, which is especially important at cloud scale. Generally, we recommend storing Azure resources metrics and diagnostics data in a Log Analytics workspace rather than in a storage account. This way, you can use Kusto queries to obtain the data you want quickly and in a structured format. You can also use Azure Monitor APIs and Azure Log Analytics APIs.
- **Log data in the production environment.** Capture robust telemetry data while the application is running in the production environment, so you have sufficient information to diagnose the cause of issues in the production state.
- **Log events at service boundaries.** Include a correlation ID that flows across service boundaries. If a transaction flows through multiple services and one of them fails, the correlation ID helps you track requests across your application and pinpoints why the transaction failed.
- **Use asynchronous logging.** Synchronous logging operations sometimes block your application code, causing requests to back up as logs are written. Use asynchronous logging to preserve availability during application logging.
- **Separate application logging from auditing.** Audit records are commonly maintained for compliance or regulatory requirements and must be complete. To avoid dropped transactions, maintain audit logs separately from diagnostic logs.

All application resources should be configured to route diagnostic logs and metrics to the chosen log aggregation technology. Azure Policy should also be used as a device to ensure the consistent use of diagnostic settings across the application, to enforce the desired configuration for each Azure service.

Application level events should be automatically correlated with resource level metrics to quantify the current application state. The overall health state can be impacted by both application level issues as well as resource level failures. Telemetry correlation should be used to ensure transactions can be mapped through the end-to-end application and critical system flows, as this is vital to root cause analysis (RCA) for failures. Platform level metrics and logs such as CPU percentage, network in/out, and disk operations/sec should be collected from the application to inform a health model and detect/predict issues. This can also help to distinguish between transient and non-transient faults.

White-box and black-box monitoring

Use white box monitoring to instrument the application with semantic logs and metrics. Application level metrics and logs, such as current memory consumption or request latency, should be collected from the application to inform a health model and detect/predict issues.

Use black-box monitoring to measure platform services and the resulting customer experience. Black box monitoring tests externally visible application behavior without knowledge of the internals of the system. This is a common approach to measuring customer-centric service level indicators (SLIs), service level objectives (SLOs), and service level agreements (SLAs).

Use critical system flows in the health model

The health model should be able to surface the respective health of critical system flows or key subsystems to ensure appropriate operational prioritization is applied. For example, the health model should be able to represent the current state of the user login transaction flow.

Create good health probes

The health and performance of an application can degrade over time, and degradation might not be noticeable until the application fails.

Implement probes or check functions, and run them regularly from outside the application. These checks can be as simple as measuring response time for the application as a whole, for individual parts of the application, for specific services that the application uses, or for separate components.

Check functions can run processes to ensure that they produce valid results, measure latency and check availability, and extract information from the system.

The HealthProbesSample sample shows how to set up health probes. It provides an Azure Resource Manager (ARM) template to set up the infrastructure. A load balancer accepts public requests and load balance to a set of virtual machines. The health probe is set up so that it can check for service's path /Health.

Next step

Best practices

Related links

- For information on monitoring metrics, see Azure Monitor Metrics overview.
- For information on using Application Insights, see What is Application Insights?

Go back to the main article: Monitoring

Monitoring best practices for reliability in Azure applications

11/19/2021 • 2 minutes to read • Edit Online

This article lists Azure best practices to enhance monitoring Azure applications for reliability. These best practices are derived from our experience with Azure reliability and the experiences of customers like yourself.

Implement these best practices for monitoring and alerts in your application so you can detect failures and alert an operator to fix them.

Implement health probes and check functions

Run them regularly from outside the application to identify degradation of application health and performance.

Check long-running workflows

Catching issues early can minimize the need to roll back the entire workflow or to execute multiple compensating transactions.

Maintain application logs

- Log applications in production and at service boundaries.
- Use semantic and asynchronous logging.
- Separate application logs from audit logs.

Measure remote call statistics

Measure remote call statistics, and share the data with the application team to give your operations team an instantaneous view into application health, summarize remote call metrics, such as latency, throughput, and errors in the 99 and 95 percentiles. Perform statistical analysis on the metrics to uncover errors that occur within each percentile.

Track transient exceptions and retries

A trend of increasing exceptions over time indicates that the service is having an issue and may fail. Track transient exceptions and retries over an appropriate time frame to prevent failure.

Set up an early warning system

Identify the key performance indicators (KPIs) of an application's health, such as transient exceptions and remote call latency, and set appropriate threshold values for each of them. Send an alert to operations when the threshold value is reached.

Operate within Azure subscription limits

Azure subscriptions have limits on certain resource types, such as the number of resource groups, cores, and storage accounts. Watch your use of resource types.

Monitor third-party services

Log your invocations and correlate them with your application's health and diagnostic logging using a unique identifier.

Train multiple operators

Train multiple operators to monitor the application and to perform manual recovery steps. Make sure there is always at least one trained operator active.

Reliability patterns

11/19/2021 • 3 minutes to read • Edit Online

Availability

Availability is measured as a percentage of uptime, and defines the proportion of time that a system is functional and working. Availability is affected by system errors, infrastructure problems, malicious attacks, and system load. Cloud applications typically provide users with a service level agreement (SLA), which means that applications must be designed and implemented to maximize availability.

PATTERN	SUMMARY
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes, to smooth intermittent heavy loads.
Throttling	Control the consumption of resources by an instance of an application, an individual tenant, or an entire service.

To mitigate against availability risks from malicious Distributed Denial of Service (DDoS) attacks, implement the native Azure DDoS protection standard service or a third party capability.

High availability

Azure infrastructure is composed of geographies, regions, and Availability Zones, which limit the blast radius of a failure and therefore limit potential impact to customer applications and data. The Azure Availability Zones construct was developed to provide a software and networking solution to protect against datacenter failures and to provide increased high availability (HA) to our customers. With HA architecture there is a balance between high resilience, low latency, and cost.

PATTERN	SUMMARY
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.

PATTERN	SUMMARY
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.

Resiliency

Resiliency is the ability of a system to gracefully handle and recover from failures, both inadvertent and malicious.

The nature of cloud hosting, where applications are often multi-tenant, use shared platform services, compete for resources and bandwidth, communicate over the Internet, and run on commodity hardware means there is an increased likelihood that both transient and more permanent faults will arise. The connected nature of the internet and the rise in sophistication and volume of attacks increase the likelihood of a security disruption.

Detecting failures and recovering quickly and efficiently, is necessary to maintain resiliency.

PATTERN	SUMMARY
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.
Compensating Transaction	Undo the work performed by a series of steps, which together define an eventually consistent operation.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Retry	Enable an application to handle anticipated, temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that's previously failed.
Scheduler Agent Supervisor	Coordinate a set of actions across a distributed set of services and other remote resources.

Overview of the security pillar

11/19/2021 • 7 minutes to read • Edit Online

Information Security has always been a complex subject, and it evolves quickly with the creative ideas and implementations of attackers and security researchers. The origin of security vulnerabilities started with identifying and exploiting common programming errors and unexpected edge cases. However over time, the attack surface that an attacker may explore and exploit has expanded well beyond that. Attackers now freely exploit vulnerabilities in system configurations, operational practices, and the social habits of the systems' users. As system complexity, connectedness, and the variety of users increase, attackers have more opportunities to identify unprotected edge cases and to "hack" systems into doing things they were not designed to do.

Security is one of the most important aspects of any architecture. It provides confidentiality, integrity, and availability assurances against deliberate attacks and abuse of your valuable data and systems. Losing these assurances can negatively impact your business operations and revenue, and your organization's reputation. For the security pillar, we'll discuss key architectural considerations and principles for security and how they apply to Azure.

The security of complex systems depends on understanding the business context, social context, and technical context. As you design your system, cover these areas:

Understanding an IT solution as it interacts with its surrounding environment holds the key to preventing unauthorized activity and to identifying anomalous behavior that may represent a security risk. Another key success factor is adopting a mindset of assuming failure of security controls so that you design compensating controls that limit risk and damage in the event a primary control fails. Assuming failures is sometimes referred to as "assume breach" or "assume compromise" and is closely related to the "Zero Trust" approach of continuously validating security assurances. The "Zero Trust" approach is described in the Security Design Principles section in more detail.

Cloud architectures can help simplify the complex task of securing an enterprise estate through specialization and shared responsibilities:

Specialization: Instead of hundreds of thousands of organizations individually developing deep expertise on managing and securing common elements like datacenter physical security, firmware patching, and hypervisor configuration, specialist teams at cloud providers can develop advanced capabilities to operate and secure the systems on behalf of these organizations. The economies of scale allow cloud provider specialist teams to invest in optimization of management and security that far exceeds the ability of most organizations.

Cloud providers must be compliant with the same IT regulatory requirements as the aggregate of all their

customers and must develop expertise to defend against the aggregate set of adversaries attacking their customers. As a consequence, the default security posture of applications deployed to the cloud is frequently much better than that of applications hosted on-premises.

Shared Responsibility Model: As computing environments move from customer-controlled datacenters to the cloud, the responsibility of security also shifts. Security of the operational environment is now a concern shared by both cloud providers and customers. By shifting these responsibilities to a cloud service like Azure, organizations can reduce focus on activities that aren't core business competencies. Depending on the specific technology choices, some security protections will be built into the particular service, while addressing others will remain the customer's responsibility. To ensure that the proper security controls are provided, a careful evaluation of the services and technology choices becomes necessary.

Shared Responsibility and Key Strategies

After reading this document, you will be equipped with key insights on how to improve the security posture of your architecture.

As part of your architecture design, you should consider all relevant areas that affect the success of your application. While this article is concerned primarily with security principles, you should also prioritize other requirements of a well-designed system including your applications' availability, scalability, costs, and operational characteristics (trading off one over the other as necessary). It is important to note that consistently sacrificing security for gains in the other areas is not advisable as security risks tend to increase dynamically over time.

This results in three key strategies:

- **Establish a modern perimeter:** For the elements that your organization controls to ensure you have a consistent set of controls (a perimeter) between those assets and the threats to them. This should be designed based on intercepting authentication requests for the resources (identity controls) vs. the traditional approach of intercepting network traffic on enterprise networks, which isn't feasible for enterprise assets outside the network.

  More on this and how it relates to Zero Trust and Enterprise Segmentation are in the Governance, Risk, and Compliance and Network Security & Containment sections.

- **Modernize infrastructure security:** For operating systems and middleware elements that are required by legacy applications; take advantage of cloud technology to reduce security risk to the organization. For example, knowing whether all servers in a physical datacenter are updated with security patches has always been extremely challenging because of discoverability. Software-defined datacenters allow easy and rapid discovery of all resources, enabling technology like Microsoft Defender for Cloud to quickly and accurately measure the patch state of all servers and remediate them.

- **"Trust but verify" each cloud provider:** For the elements which are under the control of the cloud

provider. You should ensure the security practices and regulatory compliance of each cloud provider (large and small) meet your requirements.

To assess your workload using the tenets found in the Microsoft Azure Well-Architected Framework, see the Microsoft Azure Well-Architected Review.

These are the topics we cover in the security pillar of the Microsoft Azure Well-Architected Framework

SECURITY TOPIC	DESCRIPTION
Security design principles	These principles support these three key strategies and describe a securely architected system hosted on cloud or on-premises datacenters (or a combination of both).
Governance, risk, and compliance	How is the organization's security going to be monitored, audited, and reported? What types of risks does the organization face while trying to protect identifiable information, Intellectual Property (IP), financial information? Are there specific industry, government, or regulatory requirements that dictate or provide recommendation on criteria that your organization's security controls must meet?
Regulatory compliance	Governments and other organizations frequently publish standards to help define good security practices (due diligence) so that organizations can avoid being negligent in security.
Administration	Administration is the practice of monitoring, maintaining, and operating Information Technology (IT) systems to meet service levels that the business requires. Administration introduces some of the highest impact security risks because performing these tasks requires privileged access to a very broad set of these systems and applications.
Applications and services	Applications and the data associated with them ultimately act as the primary store of business value on a cloud platform.
Identity and access management	Identity provides the basis of a large percentage of security assurances.
Information protection and storage	Protecting data at rest is required to maintain confidentiality, integrity, and availability assurances across all workloads.

SECURITY TOPIC	DESCRIPTION
Network security and containment	Network security has been the traditional linchpin of enterprise security efforts. However, cloud computing has increased the requirement for network perimeters to be more porous and many attackers have mastered the art of attacks on identity system elements (which nearly always bypass network controls).
Security Operations	Security operations maintain and restores the security assurances of the system as live adversaries attack it. The tasks of security operations are described well by the NIST Cybersecurity Framework functions of Detect, Respond, and Recover.

Identity management

Consider using Azure Active Directory (Azure AD) to authenticate and authorize users. Azure AD is a fully managed identity and access management service. You can use it to create domains that exist purely on Azure, or integrate with your on-premises Active Directory identities. Azure AD also integrates with Office365, Dynamics CRM Online, and many third-party SaaS applications. For consumer-facing applications, Azure Active Directory B2C lets users authenticate with their existing social accounts such as Facebook, Google, or LinkedIn, or create a new user account that is managed by Azure AD.

If you want to integrate an on-premises Active Directory environment with an Azure network, several approaches are possible, depending on your requirements. For more information, see our Identity Management reference architectures.

Protect your infrastructure

Control access to the Azure resources that you deploy. Every Azure subscription has a trust relationship with an Azure AD tenant. Use Azure role-based access control (Azure RBAC) to grant users within your organization the correct permissions to Azure resources. Grant access by assigning Azure roles to users or groups at a certain scope. The scope can be a subscription, a resource group, or a single resource. Audit all changes to infrastructure.

Application security

In general, the security best practices for application development still apply in the cloud. These include things like using SSL everywhere, protecting against CSRF and XSS attacks, preventing SQL injection attacks, and so on.

Cloud applications often use managed services that have access keys. Never check these keys into source control. Consider storing application secrets in Azure Key Vault.

Data sovereignty and encryption

Make sure that your data remains in the correct geopolitical zone when using Azure data services. Azure's geo-replicated storage uses the concept of a paired region in the same geopolitical region.

Use Key Vault to safeguard cryptographic keys and secrets. By using Key Vault, you can encrypt keys and secrets by using keys that are protected by hardware security modules (HSMs). Many Azure storage and DB services support data encryption at rest, including:

- Azure Storage

- Azure SQL Database
- Azure Synapse Analytics
- Cosmos DB

Security resources

- Microsoft Defender for Cloud provides integrated security monitoring and policy management for your workload.
- Azure Security Documentation
- Microsoft Trust Center

Security design principles

11/19/2021 • 4 minutes to read • Edit Online

These principles support these three key strategies and describe a securely architected system hosted on cloud or on-premises datacenters (or a combination of both). Application of these principles will dramatically increase the likelihood your security architecture will maintain assurances of confidentiality, integrity, and availability.

Each recommendation in this document includes a description of why it is recommended, which maps to one of more of these principles:

- **Align security priorities to mission:** Security resources are almost always limited, so prioritize efforts and assurances by aligning security strategy and technical controls to the business using classification of data and systems. Security resources should be focused first on people and assets (such as systems, data, and accounts) with intrinsic business value and those with administrative privileges over business critical assets.
- **Build a comprehensive strategy:** A security strategy should consider investments in culture, processes, and security controls across all system components. The strategy should also consider security for the full lifecycle of system components including the supply chain of software, hardware, and services.
- **Drive simplicity:** Complexity in systems leads to increased human confusion, errors, automation failures, and difficulty of recovering from an issue. Favor simple and consistent architectures and implementations.
- **Design for attackers:** Your security design and prioritization should be focused on the way attackers see your environment, which is often not the way IT and application teams see it. Inform your security design and test it with **penetration testing** to simulate one-time attacks. Use **red teams** to simulate long-term persistent attack groups. Design your enterprise segmentation strategy and other security controls to **contain** attacker lateral movement within your environment. Actively measure and reduce the potential **attack surface** that attackers target for exploitation of resources within the environment.
- **Use native controls:** Favor native security controls built into cloud services over external controls from third parties. Native security controls are maintained and supported by the service provider, eliminating or reducing effort required to integrate external security tooling and update those integrations over time.
- **Use identity as primary access control:** Access to resources in cloud architectures is primarily governed by identity-based authentication and authorization for access controls. Your account control strategy should rely on identity systems for controlling access rather than relying on network controls or direct use of cryptographic keys.
- **Accountability:** Designate clear ownership of assets and security responsibilities and ensure actions are traceable for nonrepudiation. You should also ensure entities have been granted the least privilege required (to a manageable level of granularity).
- **Embrace automation:** Automation of tasks decreases the chance of human error that can create risk, so both IT operations and security best practices should be automated as much as possible to reduce human errors (while ensuring skilled humans govern and audit the automation).
- **Focus on information protection:** Intellectual property is frequently one of the biggest repositories of organizational value and this data should be protected anywhere it goes including cloud services, mobile devices, workstations, or collaboration platforms (without impeding collaboration that allows for business value creation). Your security strategy should be built around classifying information and assets

to enable security prioritization, using strong access control and encryption technology, and meeting business needs like productivity, usability, and flexibility.

- **Design for resilience:** Your security strategy should assume that controls will fail and design accordingly. Making your security posture more resilient requires several approaches working together:
  - **Balanced investment:** Invest across core functions spanning the full NIST Cybersecurity Framework lifecycle (identify, protect, detect, respond, and recover) to ensure that attackers who successfully evade preventive controls lose access from detection, response, and recovery capabilities.
  - **Ongoing maintenance:** Maintain security controls and assurances to ensure that they don't decay over time with changes to the environment or neglect.
  - **Ongoing vigilance:** Ensure that anomalies and potential threats that could pose risks to the organizations are addressed in a timely manner.
  - **Defense in depth:** Consider additional controls in the design to mitigate risk to the organization in the event a primary security control fails. This design should consider how likely the primary control is to fail, the potential organizational risk if it does, and the effectiveness of the additional control (especially in the likely cases that would cause the primary control to fail).
  - **Least privilege:** This is a form of defense in depth to limit the damage that can be done by any one account. Accounts should be granted the least amount of privilege required to accomplish their assigned tasks. Restrict the access by permission level and by time. This helps mitigate the damage of an external attacker who gains access to the account and/or an internal employee who inadvertently or deliberately (for example, insider attack) compromises security assurances.
- **Baseline and benchmark:** To ensure your organization considers current thinking from outside sources, evaluate your strategy and configuration against external references (including compliance requirements). This helps to validate your approaches, minimize risk of inadvertent oversight, and the risk of punitive fines for noncompliance.
- **Drive continuous improvement:** Systems and existing practices should be regularly evaluated and improved to ensure they are and remain effective against attackers who continuously improve and the continuous digital transformation of the enterprise. This should include processes that proactively integrate learnings from real world attacks, realistic penetration testing and red team activities, and other sources as available.
- **Assume zero trust:** When evaluating access requests, all requesting users, devices, and applications should be considered untrusted until their integrity can be sufficiently validated. Access requests should be granted conditionally based on the requestor's trust level and the target resource's sensitivity. Reasonable attempts should be made to offer means to increase trust validation (for example, request multifactor authentication) and remediate known risks (change known-leaked password, remediate malware infection) to support productivity goals.
- **Educate and incentivize security:** The humans that are designing and operating the cloud workloads are part of the whole system. It is critical to ensure that these people are educated, informed, and incentivized to support the security assurance goals of the system. This is particularly important for people with accounts granted broad administrative privileges.

Governance, risk, and compliance

11/19/2021 • 4 minutes to read • Edit Online

As part of overall design, prioritize where to invest the available resources; financial, people, and time. Constraints on those resources also affect the security implementation across the organization. To achieve an appropriate ROI on security the organization needs to first understand and define its security priorities.

- **Governance:** How is the organization's security going to be monitored, audited, and reported? Design and implementation of security controls within an organization is only the beginning of the story. How does the organization know that things are actually working? Are they improving? Are there new requirements? Is there mandatory reporting? Similar to compliance there may be external industry, government or regulatory standards that need to be considered.
- **Risk:** What types of risks does the organization face while trying to protect identifiable information, Intellectual Property (IP), financial information? Who may be interested or could use this information if stolen, including external and internal threats as well as unintentional or malicious? A commonly forgotten but extremely important consideration within risk is addressing Disaster Recovery and Business Continuity.
- **Compliance:** Is there a specific industry, government, or regulatory requirements that dictate or provide recommendation on criteria that your organization's security controls must meet? Examples of such standards, organizations, controls, and legislation are ISO27001, NIST, PCI-DSS.

The collective role of organization(s) is to manage the security standards of the organization through their lifecycle:

- **Define:** Set organizational policies for operations, technologies, and configurations based on internal factors (business requirements, risks, asset evaluation) and external factors (benchmarks, regulatory standards, threat environment).
- **Improve:** Continually push these standards incrementally forward towards the ideal state to ensure continual risk reduction.
- **Sustain:** Ensure the security posture doesn't degrade naturally over time by instituting auditing and monitoring compliance with organizational standards.

Prioritize security best practices investments

Security best practices are ideally applied proactively and completely to all systems as you build your cloud program, but this isn't reality for most enterprise organizations. Business goals, project constraints, and other factors often cause organizations to balance security risk against other risks and apply a subset of best practices at any given point.

We recommend applying as many of the best practices as early as possible, and then working to retrofit any gaps over time as you mature your security program to include review, prioritization, and proactive application of best practices to cloud resources. We recommend evaluating the following considerations when prioritizing which to follow first:

- **High business impact and highly exposed systems:** These include systems with direct intrinsic value as well as the systems that provide attackers a path to them. For more information, see Identify and classify business critical applications.
- **Easiest to implement mitigations:** Identify quick wins by prioritizing the best practices, which your

organization can execute quickly because you already have the required skills, tools, and knowledge to do it (for example, implementing a Web App Firewall (WAF) to protect a legacy application). Be careful not to exclusively use (or overuse) this short-term prioritization method. Doing so can increase your risk by preventing your program from growing and leaving critical risks exposed for extended periods.

Microsoft has provided some prioritized lists of security initiatives to help organizations start with these decisions based on our experience with threats and mitigation initiatives in our own environments and across our customers. See Module 4a of the Microsoft CISO Workshop.

Checklist

What considerations for compliance and governance did you make?

- Create a landing zone for the workload. The infrastructure must have appropriate controls and be repeatable with every deployment.
- Enforce creation and deletion of services and their configuration through Azure Policies.
- Ensure consistency across the enterprise by applying policies, permissions, and tags across all subscriptions through careful implementation of root management group.
- Understand regulatory requirements and operational data that may be used for audits.
- Continuously monitor and assess the compliance of your workload. Perform regular attestations to avoid fines.
- Review and apply recommendations from Azure.
- Remediate basic vulnerabilities to keep the attacker costs high.

In this section

Follow these questions to assess the workload at a deeper level.

ASSESSMENT	DESCRIPTION
Are there any regulatory requirements for this workload?	Understand all regulatory requirements. Check the Microsoft Trust Center for the latest information, news, and best practices in security, privacy, and compliance.
Is the organization using a landing zone for this workload?	Consider the security controls placed on the infrastructure into which the workload will get deployed.
Do you have a segmentation strategy	Reference model and strategies of how the functions and teams can be segmented.
Are you using management groups as part of your segmentation strategy?	Strategies using management groups to manage resources across multiple subscriptions consistently and efficiently.
What security controls do you have in place for access to Azure infrastructure?	Guidance on reducing risk exposure in scope and time when configuring critical impact accounts such as Administrators.

Azure security benchmark

The Azure Security Benchmark includes a collection of high-impact security recommendations you can use to help secure the services you use in Azure:

> The questions in this section are aligned to these controls:

- Governance and Strategy
- Posture and vulnerability management

Reference architecture

Here are some reference architectures related to governance:

Cloud Adoption Framework enterprise-scale landing zone architecture

Next steps

Provide security assurance through identity management to authenticate and grant permission to users, partners, customers, applications, services, and other entities.

Identity and access management

Related links

Go back to the main article: Security

Regulatory compliance

11/19/2021 • 3 minutes to read • Edit Online

A workload can have regulatory requirements, which may mandate that operational data, such as application logs and metrics, remain within a certain geo-political region.

These requirements may need strict security measures that affect the overall architecture, the selection, and configuration of specific PaaS, and SaaS services. The requirements also have implications for how the workload should be operationalized.

Key points

- Make sure that all regulatory and governance requirements are known, and well understood.
- Periodically perform external and, or internal workload security audits.
- Have compliance checks as part of the workload operations.
- Use Microsoft Trust Center.

Review the requirements

Regulatory organizations frequently publish standards and updates to help define good security practices so that organizations can avoid negligence. The purpose and scope of these standards, and regulations vary. The security requirements, however, can influence the design for data protection and retention, network access, and system security.

Knowing whether your cloud resources are in compliance with standards mandated by governments or industry organizations is essential in today's globalized world.

For example, a workload that handles credit card transactions is subject to the Payment Card Industry (PCI) standard. One of the requirements prohibits access between the internet and any system component in the cardholder data environment.

To provide a restrictive environment, you can choose to do the following:

- Host the workload in different Azure compute options that supports bring your own VNet.
- Remove any internet-facing endpoints by using Private Endpoints.
- Use network security groups (NSGs) rules that define authorized inbound and outbound access.

Noncompliance can lead to fines or other business impact. Work with your regulators and carefully review the standard to understand both the intent and the literal wording of each requirement. Here are some questions that may help you understand each requirement.

- How is compliance measured?
- Who approves that the workload meets the requirements?
- Are there processes for obtaining attestations?
- What are the documentation requirements?

Suggested action

Use Microsoft Defender for Cloud to assess your current compliance score and to identify the gaps.

Learn more

Tutorial: Improve your regulatory compliance

Use the Microsoft Trust Center

Keep checking the Microsoft Trust Center for the latest information, news, and best practices in security, privacy, and compliance.

- **Data governance**. Focus on protecting information in cloud services, mobile devices, workstations, or collaboration platforms. Build the security strategy by classifying and labeling information. Use strong access control and encryption technology.
- **Compliance offerings**. Microsoft offers a comprehensive set of compliance offerings to help your organization follow national, regional, and industry-specific requirements governing the collection and use of data. For information, see Compliance offerings.
- **Compliance score**. Use Microsoft Compliance Score to assess your data protection controls on an ongoing basis. Act on the recommendations to make progress toward compliance.
- **Audit reports**. Use audit reports to stay current on the latest privacy, security, and compliance-related information for Microsoft's cloud services. See Audit Reports.
- **Shared responsibility**. The workload can be hosted on Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS), or in an on-premises datacenter. Have a clear understanding about the portions of the architecture you're responsible for versus Azure. Whatever the hosting model, the following responsibilities are always retained by you:
  - Data
  - Endpoints
  - Account
  - Access management

  For more information, reference Shared responsibility in the cloud.

Elevated security capabilities

Consider whether to use specialized security capabilities in your enterprise architecture.

Dedicated HSMs and Confidential Computing have the potential to enhance security and meet regulatory requirements, but can introduce complexity that may negatively impact your operations and efficiency.

Suggested actions

We recommend careful consideration and judicious use of these security measures as required:

- **Dedicated Hardware Security Modules (HSMs)**
  Dedicated Hardware Security Modules (HSMs) may help meet regulatory or security requirements.
- **Confidential Computing**
  Confidential Computing may help meet regulatory or security requirements.

Learn more about elevated security capabilities for Azure workloads.

Operational considerations

Regulatory requirements may influence the workload operations. For example, there might be a requirement that operational data, such as application logs and metrics, remain within a certain geo-political region.

Consider automation of deployment and maintenance tasks. Automation reduces security and compliance risk by limiting opportunity to introduce human errors during manual tasks.

Related links

Azure maintains a compliance portfolio that covers US government, industry specific, and region/country standards. For more information, reference Azure compliance offerings.

Monitor the compliance of the workload to check if the security controls are aligned to the regulatory requirements. For more information, reference Security audits.

> Go back to the main article: Governance

Next

Azure landing zone

Zero-trust landing zone in Azure

11/19/2021 • 3 minutes to read • Edit Online

A *landing zone* refers to a prepared infrastructure into which a workload gets deployed. It already has the compute, data sources, access controls, and networking components provisioned. When a workload lands on Azure, the required plumbing is ready; the workload needs to plug into it.

From a security perspective, there are several benefits. First, a landing zone offers isolation by creating segments. You can isolate assets at several layers from Azure enrollment down to a subscription that has the resources for the workload. This strategy of having resources within a boundary that is separate from other parts of the organization is an effective way of detecting and containing adversary movements.

Another benefit is consistent adoption of organizational policies. Policies govern which resources can be used and their usage limits. Policies also provide identity controls. Only authenticated and authorized entities are allowed access. This approach decouples the governance requirements from the workload requirements. It's crucial that a landing zone is handed over to the workload owner with the security guardrails deployed.

A well-architected landing zone supports the zero-trust principle. The landing zone is configured with least privileges that is in compliance with enterprise security. The workload requirements must align those requirements. For instance when design networking controls, support zero-trust by opening communication paths only when necessary and only to trusted entities.

The preceding examples are conceptually simple but the implementation can get complicated for an enterprise-scale deployment. This article provides links to articles in Cloud Adoption Framework (CAF) that describe the design considerations and best practices.

Learn more

What is the Microsoft Cloud Adoption Framework for Azure?

Increase automation with Azure Blueprints

Use Azure's native automation capabilities to increase consistency, compliance, and deployment speed for workloads. A recommended way to implement a landing zone is with Azure Blueprints and Azure Policies.

Automation of deployment and maintenance tasks reduces security and compliance risk by limiting opportunity to introduce human errors during manual tasks. This will also allow both IT Operations teams and security teams to shift their focus from repeated manual tasks to higher value tasks like enabling developers and business initiatives, protecting information, and so on.

Utilize the Azure Blueprint service to rapidly and consistently deploy application environments that are compliant with your organization's policies and external regulations. Azure Blueprint Service automates deployment of environments including Azure roles, policies, resources, such as virtual machines, networking, storage, and more. Azure Blueprints builds on Microsoft's significant investment into the Azure Resource Manager to standardize resource deployment in Azure and enable resource deployment and governance based on a desired-state approach. You can use built in configurations in Azure Blueprint, make your own, or just use Resource Manager scripts for smaller scope.

Several Security and Compliance Blueprints samples are available to use as a starting template.

Enforce policy compliance

Organizations of all sizes will have security compliance requirements. Industry, government, and internal

corporate security policies all need to be audited and enforced. Policy monitoring is critical to check that initial configurations are correct and that it continues to be compliant over time.

In Azure, you can take advantage of Azure Policy to create and manage policies that enforce compliance. Like Azure Blueprints, Azure Policies are built on the underlying Azure Resource Manager capabilities in the Azure platform (and Azure Policy can also be assigned via Azure Blueprints).

For more information on how to do this in Azure, please review Tutorial: Create and manage policies to enforce compliance.

Architecture

For information about an enterprise-scale reference architecture, see Cloud Adoption Framework enterprise-scale landing zone architecture. The architecture provides considerations in these critical design areas:

- Enterprise Agreement (EA) enrollment and Azure Active Directory tenants
- Identity and access management
- Management group and subscription organization
- Network topology and connectivity
- Management and monitoring
- Business continuity and disaster recovery
- Security, governance, and compliance
- Platform automation and DevOps

Azure services

How do you consistently deploy landing zones that follow organizational policies?

Key Azure services that can help in creating a landing zone:

- Azure Blueprints sketches a solution's design parameters based on an organization's standards, patterns, and requirements.
- Azure Resource Manager template specs stores an Azure Resource Manager template (ARM template) in Azure for later deployment.
- Azure Policy enforces organizational standards and to assess compliance at-scale.
- Azure AD and Azure role-based access control (Azure RBAC) work in conjunction to provide identity and access controls.
- Microsoft Defender for Cloud
- Microsoft Defender for Cloud

Next

Use management groups to manage resources across multiple subscriptions consistently and efficiently.

Management groups

> Back to the main article: Governance

Segmentation strategies

11/19/2021 • 4 minutes to read • Edit Online

Segmentation refers to the isolation of resources from other parts of the organization. It's an effective way of detecting and containing adversary movements.

One approach to segmentation is network isolation. This approach is not recommended because different technical teams may not be aligned with the business use cases and application workloads. One outcome of such a mismatch is complexity, as especially seen with on-premises networking, and can lead to reduced velocity, or in worse cases, broad network firewall exceptions. Although network control should be considered as a segmentation strategy, it should be part of a unified segmentation strategy.

Network security has been the traditional linchpin of enterprise security efforts. However, cloud computing has increased the requirement for network perimeters to be more porous and many attackers have mastered the art of attacks on identity system elements (which nearly always bypass network controls). These factors have increased the need to focus primarily on identity-based access controls to protect resources rather than network-based access controls.

An effective segmentation strategy will guide *all* technical teams (IT, security, applications) to consistently isolate access using networking, applications, identity, and any other access controls. The strategy should aim to:

- Minimize operational friction by aligning to business practices and applications
- Contain risk by adding cost to attackers. This is done by:
  - Isolating sensitive workloads from compromise by other assets.
  - Isolating high-exposure systems from being used as a pivot to other systems.
- Monitor operations that might lead to potential violation of the integrity of the segments (account usage, unexpected traffic).

Here are some recommendations for creating a unified strategy:

- Ensure alignment of technical teams to a single strategy based on assessing business risks.
- Establish a modern perimeter based on zero-trust principles, focused on identity, devices, applications, and other signals. This helps overcome limitations of network controls in protecting from new resources and attack types.
- Reinforce network controls for legacy applications by exploring microsegmentation strategies.
- Centralize the organizational responsibility for management and security of core networking functions such as cross-premises links, virtual networking, subnetting, and IP address schemes as well as network security elements such as virtual network appliances, encryption of cloud virtual network activity and cross-premises traffic, network-based access controls, and other traditional network security components.

Reference model

Start with this reference model and adapt it to your organization's needs. This model shows how functions, resources, and teams can be segmented.

Example segments

Consider isolating shared and individual resources as shown in the preceding image.

Core Services segment

This segment hosts shared services utilized across the organization. These shared services typically include Active Directory Domain Services, DNS/DHCP, and system management tools hosted on Azure Infrastructure as a Service (IaaS) virtual machines.

Additional segments

Other segments can contain grouped resources based on certain criteria. For instance, resources that are used by one specific workload or application might be contained in a separate segment. You may also segment or sub-segment by lifecycle stage, like *development*, *test*, and *production*. Some resources might intersect, such as applications, and can use virtual networks for lifecycle stages.

Clear lines of responsibility

These are the main functions for this reference model. Permissions for these functions are described in Team roles and responsibilities.

FUNCTION	SCOPE	RESPONSIBILITY
Policy management (Core and individual segments)	Some or all resources.	Monitor and enforce compliance with external (or internal) regulations, standards, and security policy assign appropriate permission to those roles.
Central IT operations (Core)	Across all resources.	Grant permissions to the central IT department (often the infrastructure team) to create, modify, and delete resources like virtual machines and storage.

FUNCTION	SCOPE	RESPONSIBILITY
Central networking group (Core and individual segments)	All network resources.	Centralize network management and security to reduce the potential for inconsistent strategies that create potential attacker exploitable security risks. Because all divisions of the IT and development organizations do not have the same level of network management and security knowledge and sophistication, organizations benefit from leveraging a centralized network team's expertise and tooling. Ensure consistency and avoid technical conflicts, assign network resource responsibilities to a single central networking organization. These resources should include virtual networks, subnets, Network Security Groups (NSG), and the virtual machines hosting virtual network appliances.
Resource role permissions (Core)	-	For most core services, administrative privileges required are granted through the application (Active Directory, DNS/DHCP, System Management Tools). No additional Azure resource permissions are required. If your organizational model requires these teams to manage their own VMs, storage, or other Azure resources, you can assign these permissions to those roles.
Security operations (Core and individual segments)	All resources.	Assess risk factors, identify potential mitigations, and advise organizational stakeholders who accept the risk.

FUNCTION	SCOPE	RESPONSIBILITY
IT operations (individual segments)	All resources.	Grant permission to create, modify, and delete resources. The purpose of the segment (and resulting permissions) will depend on your organization structure. • Segments with resources managed by a centralized IT organization can grant the central IT department (often the infrastructure team) permission to modify these resources. • Segments managed by independent business units or functions (such as a Human Resources IT Team) can grant those teams permission to all resources in the segment. • Segments with autonomous DevOps teams don't need to grant permissions across all resources because the resource role (below) grants permissions to application teams. For emergencies, use the service admin account (break-glass account).
Service admin (Core and individual segments)		Use the service admin role only for emergencies (and initial setup if required). Do not use this role for daily tasks.

Next steps

Start with this reference model and manage resources across multiple subscriptions consistently and efficiently with management groups.

Management groups

Establish segmentation with management groups

11/19/2021 • 3 minutes to read • Edit Online

Management groups can manage resources across multiple subscriptions consistently and efficiently. However, due to its flexibility, your design can become complex and compromise security and operations.

Support your segmentation strategy with management groups

Structure management groups into a simple design that guides the enterprise segmentation model.

Management groups offer the ability to consistently and efficiently manage resources (including multiple subscriptions as needed). However, because of their flexibility, it's possible to create an overly complex design. Complexity creates confusion and negatively impacts both operations and security (as illustrated by overly complex Organizational Unit (OU) and Group Policy Object (GPO) designs for Active Directory).

Microsoft recommends aligning the top level of management groups (MGs) into a simple enterprise segmentation strategy and limit the levels to no more than two.

In the example reference, there are enterprise-wide resources used by all segments, a set of core services that share services, additional segments for each workload.

- Root management group for enterprise-wide resources.

  Use the root management group to include identities that have the requirement to apply policies across every resource. For example, regulatory requirements, such as restrictions related to data sovereignty. This group is effective in by applying policies, permissions, tags, across all subscriptions.

  Caution

  Be careful when using the root management group because the policies can affect all resources on Azure and potentially cause downtime or other negative impacts. For considerations, see Use root management group with caution later in this article.

  For complete guidance about using management groups for an enterprise, see CAF: Management group and subscription organization.

- Management group for each workload segment.

  Use a separate management group for teams with limited scope of responsibility. This group is typically required because of organizational boundaries or regulatory requirements.

- Root or segment management group for the core set of services.

Use root management group with caution

Use the Root Management Group (MG) for enterprise consistency, but test changes carefully to minimize risk of operational disruption.

The root management group enables you to ensure consistency across the enterprise by applying policies, permissions, and tags across all subscriptions. Care must be taken when planning and implementing assignments to the root management group because this can affect every resource on Azure and potentially cause downtime or other negative impacts on productivity in the event of errors or unanticipated effects.

- **Plan carefully:** Select enterprise-wide elements to the root management group that have a clear requirement to be applied across every resource and/or low impact.

Select enterprise-wide identities that have a clear requirement to be applied across all resources. Good candidates include:

- **Regulatory requirements** with clear business risk/impact. For example, restrictions related to data sovereignty.
- **Near-zero potential negative impact.** For example, policy with audit effect, tag assignment, Azure RBAC permissions assignments that have been carefully reviewed.

Use a dedicated service principal name (SPN) to execute management group management operations, subscription management operations, and role assignment. SPN reduces the number of users who have elevated rights and follows least-privilege guidelines. Assign the **User Access Administrator** at the root management group scope (/) to grant the SPN just mentioned access at the root level. After the SPN is granted permissions, the **User Access Administrator** role can be safely removed. In this way, only the SPN is part of the **User Access Administrator** role. Assign **Contributor** permission to the SPN, which allows tenant-level operations. This permission level ensures that the SPN can be used to deploy and manage resources to any subscription within your organization.

Limit the number of Azure Policy assignments made at the root management group scope (`/`). This limitation minimizes debugging inherited policies in lower-level management groups.

Don't create any subscriptions under the root management group. This hierarchy ensures that subscriptions don't only inherit the small set of Azure policies assigned at the root-level management group, which don't represent a full set necessary for a workload.

- **Test first:** Plan, test, and validate all enterprise-wide changes on the root management group before applying (policy, tags, Azure RBAC model, and so on).
  - **Test lab:** Representative lab tenant or lab segment in production tenant.
  - **Production pilot:** This can be a segment management group or designated subset in subscription(s) management group.
- **Validate changes:** to ensure they have the desired effect.

Next steps

Administrative accounts

Administrative account security

11/19/2021 • 8 minutes to read • Edit Online

Administration is the practice of monitoring, maintaining, and operating Information Technology (IT) systems to meet service levels that the business requires. Administration introduces some of the highest impact security risks because performing these tasks requires privileged access to a very broad set of these systems and applications. Attackers know that gaining access to an account with administrative privileges can get them access to most or all of the data they would target, making the security of administration one of the most critical security areas.

As an example, Microsoft makes significant investments in protection and training of administrators for our cloud systems and IT systems:

Microsoft's recommended core strategy for administrative privileges is to use the available controls to reduce risk

Reduce risk exposure (scope and time): The principle of least privilege is best accomplished with modern controls that provide privileges on demand. This help to limit risk by limiting administrative privileges exposure by:

- **Scope:** *Just enough access (JEA)* provides only the required privileges for the administrative operation required (vs. having direct and immediate privileges to many or all systems at a time, which is almost never required).
- **Time:** *Just in time (JIT)* approaches provided the required privileged as they are needed.
- **Mitigate the remaining risks:** Use a combination of preventive and detective controls to reduce risks such as isolating administrator accounts from the most common risks phishing and general web browsing, simplifying and optimizing their workflow, increasing assurance of authentication decisions, and identifying anomalies from normal baseline behavior that can be blocked or investigated.

Microsoft has captured and documented best practices for protecting administrative accounts and published prioritized roadmaps for protecting privileged access that can be used as references for prioritizing mitigations

for accounts with privileged access.

- Securing Privileged Access (SPA) roadmap for administrators of on-premises Active Directory
- Guidance for securing administrators of Azure Active Directory

Minimize number of critical impact admins

Grant the fewest number of accounts to privileges that can have a critical business impact

Each admin account represents potential attack surface that an attacker can target, so minimizing the number of accounts with that privilege helps limit the overall organizational risk. Experience has taught us that membership of these privileged groups grows naturally over time as people change roles if membership not actively limited and managed.

We recommend an approach that reduces this attack surface risk while ensuring business continuity in case something happens to an administrator:

- Assign at least two accounts to the privileged group for business continuity
- When two or more accounts are required, provide justification for each member including the original two
- Regularly review membership & justification for each group member

Managed accounts for admins

Ensure all critical impact admins are managed by enterprise directory to follow organizational policy enforcement.

Consumer accounts such as Microsoft accounts like @Hotmail.com, @live.com, @outlook.com, don't offer sufficient security visibility and control to ensure the organization's policies and any regulatory requirements are being followed. Because Azure deployments often start small and informally before growing into enterprise-managed tenants, some consumer accounts remain as administrative accounts long afterward for example, original Azure project managers, creating blind spots, and potential risks.

Separate accounts for admins

Ensure all critical impact admins have a separate account for administrative tasks (vs the account they use for email, web browsing, and other productivity tasks).

Phishing and web browser attacks represent the most common attack vectors to compromise accounts, including administrative accounts.

Create a separate administrative account for all users that have a role requiring critical privileges. For these administrative accounts, block productivity tools like Office 365 email (remove license). If possible, block arbitrary web browsing (with proxy and/or application controls) while allowing exceptions for browsing to the Azure portal and other sites required for administrative tasks.

No standing access / just in time privileges

Avoid providing permanent "standing" access for any critical impact accounts

Permanent privileges increase business risk by increasing the time an attacker can use the account to do damage. Temporary privileges force attackers targeting an account to either work within the limited times the admin is already using the account or to initiate privilege elevation (which increases their chance of being detected and removed from the environment).

Grant privileges required only as required using one of these methods:

- **Just in time:** Enable Azure AD Privileged Identity Management (PIM) or a third party solution to require following an approval workflow to obtain privileges for critical impact accounts
- **Break glass:** For rarely used accounts, follow an emergency access process to gain access to the accounts. This is preferred for privileges that have little need for regular operational usage like members of global admin accounts.

Emergency access or 'Break Glass' accounts

Ensure you have a mechanism for obtaining administrative access in case of an emergency

While rare, sometimes extreme circumstances arise where all normal means of administrative access are unavailable.

We recommend following the instructions at Managing emergency access administrative accounts in Azure AD and ensure that security operations monitor these accounts carefully.

Admin workstation security

Ensure critical impact admins use a workstation with elevated security protections and monitoring

Attack vectors that use browsing and email like phishing are cheap and common. Isolating critical impact admins from these risks will significantly lower your risk of a major incident where one of these accounts is compromised and used to materially damage your business or mission.

Choose level of admin workstation security based on the options available at https://aka.ms/securedworkstation

- **Highly Secure Productivity Device (Enhanced Security Workstation or Specialized Workstation)**
  You can start this security journey for critical impact admins by providing them with a higher security workstation that still allows for general browsing and productivity tasks. Using this as an interim step helps ease the transition to fully isolated workstations for both the critical impact admins as well as the IT staff supporting these users and their workstations.
- **Privileged Access Workstation (Specialized Workstation or Secured Workstation)**
  These configurations represent the ideal security state for critical impact admins as they heavily restrict access to phishing, browser, and productivity application attack vectors. These workstations don't allow general internet browsing, only allow browser access to Azure portal and other administrative sites.

Critical impact admin dependencies – Account/Workstation

Carefully choose the on-premises security dependencies for critical impact accounts and their workstations

To contain the risk from a major incident on-premises spilling over to become a major compromise of cloud assets, you must eliminate or minimize the means of control that on premises resources have to critical impact accounts in the cloud. As an example, attackers who compromise the on-premises Active Directory can access and compromise cloud-based assets that rely on those accounts like resources in Azure, Amazon Web Services (AWS), ServiceNow, and so on. Attackers can also use workstations joined to those on premises domains to gain access to accounts and services managed from them.

Choose the level of isolation from on premises means of control also known as security dependencies for critical impact accounts

- **User Accounts:** Choose where to host the critical impact accounts
  - Native Azure AD Accounts -*Create Native Azure AD Accounts that are not synchronized with on-

premises active directory

  - Synchronize from On Premises Active Directory
  - Use existing accounts hosted in the on-premises active directory.

- **Workstations:** Choose how you will manage and secure the workstations used by critical admin accounts:
  - Native Cloud Management and Security (Recommended): Join workstations to Azure AD & Manage/Patch them with Intune or other cloud services. Protect and Monitor with Windows Microsoft Defender for Endpoint or another cloud service not managed by on premises based accounts.
  - Manage with Existing Systems: Join existing AD domain and use existing management/security.

Passwordless or multifactor authentication for admins

Require all critical impact admins to use passwordless authentication or multifactor authentication (MFA).

Attack methods have evolved to the point where passwords alone cannot reliably protect an account. This is well documented in a Microsoft Ignite Session.

Administrative accounts and all critical accounts should use one of the following methods of authentication. These capabilities are listed in preference order by highest cost/difficulty to attack (strongest/preferred options) to lowest cost/difficult to attack:

- **Passwordless (such as Windows Hello)**
- **Passwordless (Authenticator App)**
- **Multifactor Authentication**

Note that SMS Text Message based MFA has become very inexpensive for attackers to bypass, so we recommend you avoid relying on it. This option is still stronger than passwords alone, but is much weaker than other MFA options

Enforce conditional access for admins - Zero Trust

Authentication for all admins and other critical impact accounts should include measurement and enforcement of key security attributes to support a Zero Trust strategy.

Attackers compromising Azure Admin accounts can cause significant harm. Conditional Access can significantly reduce that risk by enforcing security hygiene before allowing access to Azure management.

Configure Conditional Access policy for Azure management that meets your organization's risk appetite and operational needs.

- Require Multifactor Authentication and/or connection from designated work network
- Require Device **integrity with Microsoft Defender for Endpoint** (Strong Assurance)

Avoid granular and custom permissions

Avoid permissions that specifically reference individual resources or users

Specific permissions create unneeded complexity and confusion as they don't carry the intention to new similar resources. This then accumulates into a complex legacy configuration that is difficult to maintain or change without fear of "breaking something" – negatively impacting both security and solution agility.

Instead of assigning specific resource-specific permissions, use either

- Management Groups for enterprise-wide permissions
- Resource groups for permissions within subscriptions

Instead of granting permissions to specific users, assign access to groups in Azure AD. If there isn't an appropriate group, work with the identity team to create one. This allows you to add and remove group members externally to Azure and ensure permissions are current, while also allowing the group to be used for other purposes such as mailing lists.

Use built-in roles

Use built-in roles for assigning permissions where possible.

Customization leads to complexity that increases confusion and makes automation more complex, challenging, and fragile. These factors all negatively impact security

We recommend that you evaluate the built-in roles designed to cover most normal scenarios. Custom roles are a powerful and sometimes useful capability, but they should be reserved for cases when built in roles won't work.

Establish lifecycle management for critical impact accounts

Ensure you have a process for disabling or deleting administrative accounts when admin personnel leave the organization (or leave administrative positions)

See Regularly review critical access for more details.

Attack simulation for critical impact accounts

Regularly simulate attacks against administrative users with current attack techniques to educate and empower them.

People are a critical part of your defense, especially your personnel with access to critical impact accounts. Ensuring these users (and ideally all users) have the knowledge and skills to avoid and resist attacks will reduce your overall organizational risk.

You can use Office 365 Attack Simulation capabilities or any number of third party offerings.

Azure identity and access management considerations

11/19/2021 • 2 minutes to read • Edit Online

Most architectures have shared services that are hosted and accessed across networks. Those services share common infrastructure and users need to access resources and data from anywhere. For such architectures, a common way to secure resources is to use network controls. However, that isn't enough.

Provide security assurance through *identity management*: the process of authenticating and authorizing security principals. Use identity management services to authenticate and grant permission to users, partners, customers, applications, services, and other entities.

Checklist

How are you managing the identity for your workload?

- Define clear lines of responsibility and separation of duties for each function. Restrict access based on a need-to-know basis and least privilege security principles.
- Assign permissions to users, groups, and applications at a certain scope through Azure RBAC. Use built-in roles when possible.
- Prevent deletion or modification of a resource, resource group, or subscription through management locks.
- Use Managed Identities to access resources in Azure.
- Support a single enterprise directory. Keep the cloud and on-premises directories synchronized, except for critical-impact accounts.
- Set up Azure AD Conditional Access. Enforce and measure key security attributes when authenticating all users, especially for critical-impact accounts.
- Have a separate identity source for non-employees.
- Preferably use passwordless methods or opt for modern password methods.
- Block legacy protocols and authentication methods.

Azure security benchmark

The Azure Security Benchmark includes a collection of high-impact security recommendations you can use to help secure the services you use in Azure:

> The questions in this section are aligned to the Azure Security Benchmarks Identity and Access Control.

Azure services for identity

The considerations and best practices in this section are based on these Azure services:

- Azure AD
- Azure AD B2B
- Azure AD B2C

Reference architecture

Here are some reference architectures related to identity and access management:

Integrate on-premises AD domains with Azure AD

Integrate on-premises AD with Azure

Next steps

Monitor the communication between segments. Use data to identify anomalies, set alerts, or block traffic to mitigate the risk of attackers crossing segmentation boundaries.

Network-related risks

Related links

Five steps to securing your identity infrastructure

> Go back to the main article: Security

Roles, responsibilities, and permissions

11/19/2021 • 9 minutes to read • Edit Online

In an organization, several teams work together to make sure that the workload and the supporting infrastructure are secure. To avoid confusion that can create security risks, define clear lines of responsibility and separation of duties.

Based on Microsoft's experience with many cloud adoption projects, establishing clearly defined roles and responsibilities for specific functions in Azure will avoid confusion that can lead to human and automation errors creating security risk.

Clear lines of responsibility

Do the teams have a clear view on responsibilities and individual/group access levels?

Designate the parties responsible for specific functions in Azure.

Clearly documenting and sharing the contacts responsible for each of these functions will create consistency and facilitate communication. Based on our experience with many cloud adoption projects, this will avoid confusion that can lead to human and automation errors that create security risk.

Designate groups (or individual roles) that will be responsible for key functions.

GROUP OR INDIVIDUAL ROLE	RESPONSIBILITY
Network Security	*Typically existing network security team.* Configuration and maintenance of Azure Firewall, Network Virtual Appliances (and associated routing), Web Application Firewall (WAF), Network Security Groups, Application Security Groups (ASG), and other cross-network traffic.
Network Management	*Typically existing network operations team.* Enterprise-wide virtual network and subnet allocation.
Server Endpoint Security	*Typically IT operations, security, or jointly.* Monitor and remediate server security (patching, configuration, endpoint security).
Incident Monitoring and Response	*Typically security operations team.* Incident monitoring and response to investigate and remediate security incidents in Security Information and Event Management (SIEM) or source console such as Microsoft Defender for Cloud Azure AD Identity Protection.
Policy Management	*Typically GRC team + Architecture.* Apply governance based on risk analysis and compliance requirements. Set direction for use of Azure role-based access control (Azure RBAC), Microsoft Defender for Cloud, Administrator protection strategy, and Azure Policy to govern Azure resources.

GROUP OR INDIVIDUAL ROLE	RESPONSIBILITY
Identity Security and Standards	*Typically Security Team + Identity Team jointly.* Set direction for Azure AD directories, PIM/PAM usage, MFA, password/synchronization configuration, Application Identity Standards.

> **NOTE**
> Application roles and responsibilities should cover different access level of each operational function. For example, publish production release, access customer data, manipulate database records, and so on. Application teams should include central functions listed in the preceding table.

Assign permissions

Grant roles the appropriate permissions that start with least privilege and add more based on your operational needs. Provide clear guidance to your technical teams that implement permissions. This clarity makes it easier to detect and correct that reduces human errors such as overpermissioning.

- Assign permissions at management group for the segment rather than the individual subscriptions. This will drive consistency and ensure application to future subscriptions. In general, avoid granular and custom permissions.
- Consider the built-in roles in Azure before creating custom roles to grant the appropriate permissions to VMs and other objects.
- **Security managers** group membership may be appropriate for smaller teams/organizations where security teams have extensive operational responsibilities.

When assigning permissions for a segment, consider consistency while allowing flexibility to accommodate several organizational models. These models can range from a single centralized IT group to mostly independent IT and DevOps teams.

Reference model example

This section uses this Reference model to demonstrate the considerations for assigning permissions for different segments. Microsoft recommends starting from these models and adapting to your organization.

Core services reference permissions

This segment hosts shared services utilized across the organization. These shared services typically include Active Directory Domain Services, DNS/DHCP, System Management Tools hosted on Azure Infrastructure as a Service (IaaS) virtual machines.

Security Visibility across all resources: For security teams, grant read-only access to security attributes for all technical environments. This access level is needed to assess risk factors, identify potential mitigations, and advise organizational stakeholders who accept the risk. See Security Team Visibility for more details.

Policy management across some or all resources: To monitor and enforce compliance with external (or internal) regulations, standards, and security policy, assign appropriate permission to those roles. The roles and permissions you choose will depend on the organizational culture and expectations of the policy program. See Microsoft Cloud Adoption Framework for Azure.

Before defining the policies, consider:

- How is the organization's security audited and reported? Is there mandatory reporting?
- Are the existing security practices working?
- Are there any requirements specific to industry, government, or regulatory requirements?

Designate group(s) (or individual roles) for central functions that affect shared services and applications.

After the policies are set, continuously improve those standards incrementally. Make sure that the security posture doesn't degrade over time by having auditing and monitoring compliance. For information about managing security standards of an organization, see governance, risk, and compliance (GRC).

Central IT operations across all resources: Grant permissions to the central IT department (often the infrastructure team) to create, modify, and delete resources like virtual machines and storage. **Contributor** or **Owner** roles are appropriate for this function.

Central networking group across network resources: To ensure consistency and avoid technical conflicts, assign network resource responsibilities to a single central networking organization. These resources should include virtual networks, subnets, Network Security Groups (NSG), and the virtual machines hosting virtual network appliances. Assign network resource responsibilities to a single central networking organization. The **Network Contributor** role is appropriate for this group. See Centralize Network Management And Security for more details

Resource Role Permissions: For most core services, administrative privileges required to manage them are granted through the application (Active Directory, DNS/DHCP, System Management Tools), so no additional Azure resource permissions are required. If your organizational model requires these teams to manage their own VMs, storage, or other Azure resources, you can assign these permissions to those roles.

Workload segments with autonomous DevOps teams will manage the resources associated with each application. The actual roles and their permissions depend on the application size and complexity, the application team size and complexity, and the culture of the organization and application team.

Service admin (Break Glass Account): Use the **Service Administrator** role only for emergencies and initial setup. Do not use this role for daily tasks. See Emergency Access ('Break Glass' Accounts) for more details.

Segment reference permissions

This segment permission design provides consistency while allowing flexibility to accommodate the range of organizational models from a single centralized IT group to mostly independent IT and DevOps teams.

Security visibility across all resources: For security teams, grant read-only access to security attributes for all technical environments. This access level is needed to assess risk factors, identify potential mitigations, and advise organizational stakeholders who accept the risk. See Security Team Visibility.

Policy management across some or all resources: To monitor and enforce compliance with external (or internal) regulations, standards, and security policy assign appropriate permission to those roles. The roles and permissions you choose will depend on the organizational culture and expectations of the policy program. See Microsoft Cloud Adoption Framework for Azure.

IT Operations across all resources: Grant permission to create, modify, and delete resources. The purpose of the segment (and resulting permissions) will depend on your organization structure.

- Segments with resources managed by a centralized IT organization can grant the central IT department (often the infrastructure team) permission to modify these resources.
- Segments managed by independent business units or functions (such as a Human Resources IT Team) can grant those teams permission to all resources in the segment.
- Segments with autonomous DevOps teams don't need to grant permissions across all resources because the resource role (below) grants permissions to application teams. For emergencies, use the service admin account (break-glass account).

Central networking group across network resources: To ensure consistency and avoid technical conflicts, assign network resource responsibilities to a single central networking organization. These resources should include virtual networks, subnets, Network Security Groups (NSG), and the virtual machines hosting virtual network appliances. See Centralize Network Management And Security.

Resource Role Permissions: Segments with autonomous DevOps teams will manage the resources associated with each application. The actual roles and their permissions depend on the application size and complexity, the application team size and complexity, and the culture of the organization and application team.

Service Admin (Break Glass Account): Use the service admin role only for emergencies (and initial setup if required). Do not use this role for daily tasks. See Emergency Access ('Break Glass' Accounts) for more details.

Security team visibility

An application team needs to be aware of security initiatives to align their security improvement plans with the outcome of those activities. Provide security teams read-only access to the security aspects of all technical resources in their purview.

Security organizations require visibility into the technical environment to perform their duties of assessing and reporting on organizational risk. Without this visibility, security will have to rely on information provided from groups, operating the environment, which have potential conflict of interest (and other priorities).

Note that security teams may separately be granted additional privileges if they have operational responsibilities or a requirement to enforce compliance on Azure resources.

For example in Azure, assign security teams to the **Security Readers** permission that provides access to measure security risk (without providing access to the data itself).

For enterprise security groups with broad responsibility for security of Azure, you can assign this permission using:

- *Root management group* – for teams responsible for assessing and reporting risk on all resources
- *Segment management group(s)* – for teams with limited scope of responsibility (typically required because of organizational boundaries or regulatory requirements)

> **IMPORTANT**
>
> Because security will have broad access to the environment (and visibility into potentially exploitable vulnerabilities), treat security teams as critical impact accounts and apply the same protections as administrators. The Administration section details these controls for Azure.

Suggested actions

- Define a process for aligning communication, investigation, and hunting activities with the application team.
- Following the principle of least privilege, establish access control to all cloud environment resources for security teams with sufficient access to gain required visibility into the technical environment and to perform their duties of assessing, and reporting on organizational risk.

Learn more

Engage your organization's security team

Manage connected tenants

Does your security team have visibility into all existing subscriptions and cloud environments? How do they discover new ones?

Ensure your security organization is aware of all enrollments and associated subscriptions connected to your existing environment (via ExpressRoute or Site-Site VPN) and monitoring as part of the overall enterprise.

These Azure resources are part of your enterprise environment and security organizations require visibility into

them. Security organizations need this access to assess risk and to identify whether organizational policies and applicable regulatory requirements are being followed.

The organizations' cloud infrastructure should be well documented, with security team access to all resources required for monitoring and insight. Frequent scans of the cloud-connected assets should be performed to ensure no additional subscriptions or tenants have been added outside of organizational controls. Regularly review Microsoft guidance to ensure security team access best practices are consulted and followed.

Suggested actions

Ensure all Azure environments that connect to your production environment and network apply your organization's policy, and IT governance controls for security.

You can discover existing connected tenants using a tool provided by Microsoft. Guidance on permissions

Next steps

Restrict access to Azure resources based on a need-to-know basis starting with the principle of least privilege security and add more based on your operational needs.

Azure control plane security

Related links

For considerations about using management groups to reflect the organization's structure within an Azure Active Directory (Azure AD) tenant, see CAF: Management group and subscription organization.

Back to the main article: Azure identity and access management considerations

Azure control plane security

11/19/2021 • 3 minutes to read • Edit Online

The term *control plane* refers to the management of resources in your subscription. These activities include creating, updating, and deleting Azure resources as required by the technical team.

Azure Resource Manager handles all control plane requests and applies restrictions that you specify through Azure role-based access control (Azure RBAC), Azure Policy, locks. Apply those restrictions based on the requirement of the organization.

It's recommended to implement Infrastructure as Code, and to deploy application infrastructure through automation, and CI/CD for consistency and auditing purposes.

Key points

- Restrict access based on a need-to-know basis and least privilege security principles.
- Assign permissions to users, groups, and applications at a certain scope through Azure RBAC.
- Use built-in roles when possible.
- Prevent deletion or modification of a resource, resource group, or subscription through management locks.
- Use less critical control in your CI/CD pipeline for development and test environments.

Roles and permission assignment

Is the workload infrastructure protected with Azure role-based access control (Azure RBAC)?

Azure role-based access control (Azure RBAC) provides the necessary tools to maintain separation of concerns for administration and access to application infrastructure. Decide who has access to resources at the granular level and what they can do with those resources. For example:

- Developers can't access production infrastructure.
- Only the SecOps team can read and manage Key Vault secrets.
- If there are multiple teams, Project A team can access and manage Resource Group A and all resources within.

> Grant roles the appropriate permissions that start with least privilege and add more based on your operational needs. Provide clear guidance to your technical teams that implement permissions. This clarity makes it easier to detect and correct which reduces human errors such as overpermissioning.

Azure RBAC helps you manage that separation. You can assign permissions to users, groups, and applications at a certain scope. The scope of a role assignment can be a subscription, a resource group, or a single resource. For details, see Azure role-based access control (Azure RBAC).

- Assign permissions at management group instead of individual subscriptions to drive consistency and ensure application to future subscriptions.
- Consider the built-in roles before creating custom roles to grant the appropriate permissions to resources and other objects.

For example, assign security teams with the **Security Readers** permission that provides access needed to assess risk factors, identify potential mitigations, without providing access to the data.

> **IMPORTANT**
> Treat security teams as critical accounts and apply the same protections as administrators.

Learn more

Azure RBAC documentation

Management locks

Are there resource locks applied on critical parts of the infrastructure?

Unlike Azure role-based access control, management locks are used to apply a restriction across all users and roles.

Critical infrastructure typically doesn't change often. Use management locks to prevent deletion or modification of a resource, resource group, or subscription. Lock in use cases where only specific roles and users with permissions can delete, or modify resources.

As an administrator, you may need to lock a subscription, resource group, or resource to prevent other users in your organization from accidentally deleting or modifying critical resources. You can set the lock level to `CanNotDelete` or `ReadOnly`. In the portal, the locks are called **Delete** and **Read-only**, respectively:

- *CanNotDelete* means authorized users can still read and modify a resource, but they can't delete the resource.
- *ReadOnly* means authorized users can read a resource, but they can't delete or update the resource. Applying this lock is similar to restricting all authorized users to the permissions granted by the *Reader* role.

When you apply a lock at a parent scope, all resources within that scope inherit the same lock. Even resources you add later inherit the lock from the parent. The most restrictive lock in the inheritance takes precedence.

Unlike role-based access control, you use management locks to apply a restriction across all users and roles. To learn about setting permissions for users and roles, see Azure role-based access control (Azure RBAC).

Identify critical infrastructure and evaluate resource lock suitability.

Set locks in the DevOps process carefully because modification locks can sometimes block automation. For examples of those blocks and considerations, see Considerations before applying locks.

Suggested actions

- Restrict application infrastructure access to CI/CD only.
- Use conditional access policies to restrict access to Microsoft Azure Management.
- Configure role-based and resource-based authorization within Azure AD.

Learn more

- Manage access to Azure management with Conditional Access
- Role-based and resource-based authorization

Next steps

Grant or deny access to a system by verifying whether the accessor has the permissions to perform the requested action.

Authentication

Related links

Back to the main article: Azure identity and access management considerations

Authentication with Azure AD

11/19/2021 • 11 minutes to read • Edit Online

Authentication is a process that grants or denies access to a system by verifying the accessor's identity. Use a managed identity service for all resources to simplify overall management (such as password policies) and minimize the risk of oversights or human errors. Azure Active Directory (Azure AD) is the one-stop-shop for identity and access management service for Azure.

Key points

- Use Managed Identities to access resources in Azure.
- Keep the cloud and on-premises directories synchronized, except for high-privilege accounts.
- Preferably use passwordless methods or opt for modern password methods.
- Enable Azure AD conditional access based on key security attributes when authenticating all users, especially for high-privilege accounts.

Use identity-based authentication

How is the application authenticated when communicating with Azure platform services?

Managed identities enable Azure Services to authenticate to each other without presenting explicit credentials via code and increase security.

Managed identities for Azure resources is a feature of Azure Active Directory. Each of the Azure services that support managed identities for Azure resources are subject to their own timeline. Make sure you review the availability status of managed identities for your resource and known issues before you begin. The feature provides Azure services with an automatically managed identity in Azure AD. You can use the identity to authenticate to any service that supports Azure AD authentication, including Key Vault, without any credentials in your code. The managed identities for Azure resources feature is free with Azure AD for Azure subscriptions, there's no additional cost.

There are two types of managed identities:

- A system-assigned managed identity is enabled directly on an Azure service instance. When the identity is enabled, Azure creates an identity for the instance in the Azure AD tenant that's trusted by the subscription of the instance. After the identity is created, the credentials are provisioned onto the instance. The life cycle of a system-assigned identity is directly tied to the Azure service instance that it's enabled on. If the instance is deleted, Azure automatically cleans up the credentials and the identity in Azure AD.
- A user-assigned managed identity is created as a standalone Azure resource. Through a create process, Azure creates an identity in the Azure AD tenant that's trusted by the subscription in use. After the identity is created, the identity can be assigned to one or more Azure service instances. The life cycle of a user-assigned identity is managed separately from the life cycle of the Azure service instances to which it's assigned.

Authenticate with identity services instead of cryptographic keys. On Azure, Managed Identities eliminate the need to store credentials that might be leaked inadvertently. When Managed Identity is enabled for an Azure resource, it's assigned an identity that you can use to obtain Azure AD tokens. For more information, see Azure AD-managed identities for Azure resources.

For example, an Azure Kubernetes Service (AKS) cluster needs to pull images from Azure Container Registry (ACR). To access the image, the cluster needs to know the ACR credentials. The recommended way is to enable Managed Identities during cluster configuration. That configuration assigns an identity to the cluster and allows

it to obtain Azure AD tokens.

This approach is secure because Azure handles the management of the underlying credentials for you.

- The identity is tied to the lifecycle of the resource, in the AKS cluster example. When the resource is deleted, Azure automatically deletes the identity.
- Azure AD manages the timely rotation of secrets for you.

> **TIP**
>
> Here are the resources for the preceding example:
>
> GitHub: Azure Kubernetes Service (AKS) Secure Baseline Reference Implementation.
>
> The design considerations are described in Azure Kubernetes Service (AKS) production baseline.

Suggested actions

- Review workload authentication and identify opportunities to convert explicit credentials (for example, connection string and API key) to use managed identities.
- For all new Azure workloads, standardize on using managed identities where applicable.

Learn more

What are managed identities for Azure resources?

What kind of authentication is required by application APIs?

Don't assume that API URLs used by a workload are hidden and can't get exposed to attackers. For example, JavaScript code on a website can be viewed. A mobile application can be decompiled and inspected. Even for internal APIs used only on the backend, a requirement of authentication can increase the difficulty of lateral movement if an attacker gets network access. Typical mechanisms include API keys, authorization tokens, IP restrictions.

Managed Identity can help an API be more secure because it replaces the use of human-managed service principals and can request authorization tokens.

How is user authentication handled in the application?

Don't use custom implementations to manage user credentials. Instead, use Azure AD or other managed identity providers such as Microsoft account Azure B2C. Managed identity providers provide additional security features such as modern password protections, multifactor authentication (MFA), and resets. In general, passwordless protections are preferred. Also, modern protocols like OAuth 2.0 use token-based authentication with limited timespan.

Are authentication tokens cached securely and encrypted when sharing across web servers?

Application code should first try to get OAuth access tokens silently from a cache before attempting to acquire a token from the identity provider, to optimize performance and maximize availability. Tokens should be stored securely and handled as any other credentials. When there's a need to share tokens across application servers (instead of each server acquiring and caching their own) encryption should be used.

For information, see Acquire and cache tokens.

Choose a system with cross-platform support

Use a single identity provider for authentication on all platforms (operating systems, cloud providers, and third-party services.

Azure AD can be used to authenticate Windows, Linux, Azure, Office 365, other cloud providers, and third-party services as service providers.

For example, improve the security of Linux virtual machines (VMs) in Azure with Azure AD integration. For details, see Log in to a Linux virtual machine in Azure using Azure Active Directory authentication.

Centralize all identity systems

Keep your cloud identity synchronized with the existing identity systems to ensure consistency and reduce human errors.

Consistency of identities across cloud and on-premises will reduce human error and resulting security risk. Teams managing resources in both environments need a consistent authoritative source to achieve security assurances. For monitoring, if identity can be determined without an intermediate mapping process, security efficiency is improved.

Synchronization is all about providing users an identity in the cloud based on their on-premises identity. Whether or not they will use synchronized account for authentication or federated authentication, the users will still need to have an identity in the cloud. This identity will need to be maintained and updated periodically. The updates can take many forms, from title changes to password changes.

Start by evaluating the organization's on-premises identity solution and user requirements. This evaluation is important, as it defines the technical requirements for how user identities will be created and maintained in the cloud. For the majority of organizations, Active Directory is established on-premises and will be the on-premises directory from which users will be synchronized, but this is not always the case.

Consider using Azure AD Connect for synchronizing Azure AD with your existing on-premises directory. For migration projects, have a requirement to complete this task before an Azure migration and development projects begin.

> **IMPORTANT**
>
> Don't synchronize high-privilege accounts to an on-premises directory. If an attacker gets full control of on-premises assets, they can compromise a cloud account. This strategy will limit the scope of an incident. For more information, see Critical impact account dependencies.
>
> Synchronization is blocked by default in the default Azure AD Connect configuration. Make sure that you haven't customized this configuration. For information about filtering in Azure AD, see Azure AD Connect sync: Configure filtering.

For more information, see hybrid identity providers.

> **TIP**
>
> Here are the resources for the preceding example::
>
> The design considerations are described in Integrate on-premises Active Directory domains with Azure AD.

Learn more

Synchronize the hybrid identity systems

Use passwordless authentication

Attackers constantly scan public cloud IP ranges for open management ports. They attempt to exploit weak

credentials (*password spray*) and unpatched vulnerabilities in management protocols like SSH, and RDP. Preventing direct internet access to virtual machines stops a misconfiguration or oversight becoming more serious.

Attack methods have evolved to the point where passwords alone cannot reliably protect an account. Modern authentication solutions including passwordless and multifactor authentication increase security posture through strong authentication.

Remove the use of passwords, when possible. Also, require the same set of credentials to sign in and access the resources on-premises or in the cloud. This requirement is crucial for accounts that require passwords, such as admin accounts.

With modern authentication and security features in Azure AD, that basic password should be supplemented or replaced with more secure authentication methods. Each organization has different needs when it comes to authentication. Microsoft offers the following three passwordless authentication options that integrate with Azure Active Directory (Azure AD):

- Windows Hello for Business
- Microsoft Authenticator app
- FIDO2 security keys

It's recommended to follow a four-stage plan to become passwordless:

- Develop password replacement offering
- Reduce user-visible password surface area
- Transition into password-less deployment
- Eliminate passwords from the identity directory

The following methods of authentication are ordered by highest cost/difficulty to attack (strongest/preferred options) to lowest cost/difficult to attack:

- Passwordless authentication. Some examples of this method include Windows Hello or Authenticator App.
- MFA. Although this method is more effective than passwords, we recommend that you avoid relying on SMS text message-based MFA. For more information, see Enable per-user Azure Active Directory MFA to secure sign-in events.
- Managed Identities. See Use identity-based authentication.

Those methods apply to all users, but should be applied first and strongest to accounts with administrative privileges.

An implementation of this strategy is enabling single sign-on (SSO) to devices, apps, and services. By signing in once using a single user account, you can grant access to all the applications and resources per business needs. Users don't have to manage multiple sets of usernames and passwords. You can provision or de-provision application access automatically. For more information, see Single sign-on.

Suggested actions

- Develop a passwordless strategy that requires MFA for all users without significantly impacting operations.
- Ensure policy and processes require restricting, and monitoring direct internet connectivity by virtual machines.

Learn more

- Passwordless Strategy
- Remove Virtual Machine (VM) direct internet connectivity

Use modern password protection

Require modern protections through methods that reduce the use of passwords. Modern authentication protocols support strong controls such as MFA and should be used instead of legacy authentication methods. Use of legacy methods increases risk of credential exposure.

Modern authentication is a method of identity management that offers more secure user authentication and authorization. It's available for Office 365 hybrid deployments of Skype for Business server on-premises and Exchange server on-premises, and split-domain Skype for Business hybrids.

Modern authentication is an umbrella term for a combination of authentication and authorization methods between a client (for example, your laptop or your phone) and a server, as well as some security measures that rely on access policies that you may already be familiar with. It includes:

- *Authentication methods*: MFA; smart card authentication; client certificate-based authentication
- *Authorization methods*: Microsoft's implementation of Open Authorization (OAuth)
- *Conditional access policies*: Mobile Application Management (MAM) and Azure Active Directory (Azure AD) Conditional Access

Review workloads that do not leverage modern authentication protocols and convert where possible. In addition, standardize using modern authentication protocols for all future workloads.

For Azure, enable protections in Azure AD:

1. Configure Azure AD Connect to synchronize password hashes. For information, see Implement password hash synchronization with Azure AD Connect sync.

2. Choose whether to automatically or manually remediate issues found in a report. For more information, see Monitor identity risks.

For more information about supporting modern passwords in Azure AD, see the following articles:

- What is Identity Protection?
- Enforce on-premises Azure AD Password Protection for Active Directory Domain Services
- Users at risk security report
- Risky sign-ins security report

For more information about supporting modern passwords in Office 365, see the following article:

What is modern authentication?

Enable conditional access

Modern cloud-based applications are typically accessible over the internet, making network location-based access inflexible and single-factor passwords a liability. Conditional access describes your authentication policy for an access decision. For example, if a user is connecting from an InTune-managed corporate PC, they might not be challenged for MFA every time, but if the user suddenly connects from a different device in a different geography, MFA is required.

Grant access requests based on the requestors' trust level and the target resources' sensitivity.

Are there any conditional access requirements for the application?

Workloads can be exposed over public internet and location-based network controls are not applicable. To enable conditional access, understand what restrictions are required for the use case. For example, MFA is a necessity for remote access; IP-based filtering can be used to enable adhoc debugging (VPNs are preferred).

Configure Azure AD Conditional Access by setting up Access policy for Azure management based on your

operational needs. For information, see Manage access to Azure management with Conditional Access.

Conditional access can be an effective way to phase out legacy authentication and associated protocols. The policies must be enforced for all admins and other critical impact accounts. Start by using metrics and logs to determine users who still authenticate with old clients. Next, disable any down-level protocols that aren't used, and set up conditional access for all users who aren't using legacy protocols. Finally, give notice and guidance to users about upgrading before blocking legacy authentication completely. For more information, see Azure AD Conditional Access support for blocking legacy auth.

Suggested actions

Implement conditional access policies for this workload.

Learn more about Azure AD Conditional Access.

Next

Grant or deny access to a system by verifying the accessor's identity.

Authorization

Related links

> Back to the main article: Azure identity and access management considerations

Authorization with Azure AD

11/19/2021 • 5 minutes to read • Edit Online

Authorization is a process that grants or denies access to a system by verifying whether the accessor has the permissions to perform the requested action. The accessor in this context is the workload (cloud application) or the user of the workload. The action might be operational or related to resource management. There are two main approaches to authorization: role-based and resource-based. Both can be configured with Azure AD.

Key points

- Use a mix of role-based and resource-based authorization. Start with the principle of least privilege and add more actions based on your needs.
- Define clear lines of responsibility and separation of duties for application roles and the resources it can manage. Consider the access levels of each operational function, such as permissions needed to publish production release, access customer data, manipulate database records.
- Do not provide permanent access for any critical accounts. Elevate access permissions that are based on approval and is time bound using Azure AD Privileged Identity Management (Azure AD PIM).|

Role-based authorization

This approach authorizes an action based on the role assigned to a user. For example, some actions require an administrator role.

A role is a set of permissions. For example, the administrator role has permissions to perform all read, write, and delete operations. Also, the role has a scope. The scope specifies the management groups, subscriptions, or resource groups within which the role is allowed to operate.

Applying consistent permissions to resources via management groups or resource groups reduces proliferation of custom, specific, per-resource permissions. Custom resource-based permissions are often unnecessary, and can cause confusion because they do not carry their intent to new similar resources. This process can accumulate into a complex legacy configuration that is difficult to maintain or change without fear of *breaking something*, and negatively impacting both security, and solution agility.

When assigning a role to a user consider what actions the role can perform and what is the scope of those operations. Here are some considerations for role assignment:

- Use built-in roles before creating custom roles to grant the appropriate permissions to VMs and other objects. You can assign built-in roles to users, groups, service principals, and managed identities. For more information, see Azure built-in roles.

- If you need to create custom roles, grant roles with the appropriate action. Actions are categorized into operational and data actions. To avoid overpermissioning, start with actions that have least privilege and add more based your operational or data access needs. Provide clear guidance to your technical teams that implement permissions. For more information, see Azure custom roles.

- If you have a segmentation strategy, assign permissions with a scope. For example, if you use management group to support your strategy, set the scope to the group rather than the individual subscriptions. This will drive consistency and ensure application to future subscriptions. When assigning permissions for a segment, consider consistency while allowing flexibility to accommodate several organizational models. These models can range from a single centralized IT group to mostly independent IT and DevOps teams. For information about assigning scope, see AssignableScopes.

- You can use security groups to assign permissions. However, there are disadvantages. It can get complex because the workload needs to keep track of which security groups correspond to which application roles, for each tenant. Also, access tokens can grow significantly and Azure AD includes an "overage" claim to limit the token size. See Microsoft identity platform access tokens.
- Instead of granting permissions to specific users, assign access to Azure AD groups. In addition, build a comprehensive delegation model that includes management groups, subscription, or resource groups RBAC. For more information, see Azure role-based access control (Azure RBAC).

For information about implementing role-based authorization in an ASP.NET application, see Role-based authorization.

Learn more

- Avoid granular and custom permissions
- Delegate administration in Azure AD

Resource-based authorization

With role-based authorization, a user gets the same level of control on a resource based on the user's role. However, there might be situations where you need to define access rights per resource. For example, in a resource group, you want to allow some users to delete the resource; other users cannot. In such situations, use resource-based authorization that authorizes an action based on a particular resource. Every resource has an Owner. Owner can delete the resource. Contributors can read and update but can't delete it.

> **NOTE**
> The *owner* and *contributor* roles for a resource are not the same as application roles.

You'll need to implement custom logic for resource-based authorization. That logic might be a mapping of resources, Azure AD object (like role, group, user), and permissions.

For information and code sample about implementing resource-based authorization in an ASP.NET application, see Resource-based authorization.

Authorization for critical accounts

There might be cases when you need to do activities that require access to important resources. Those resources might already be accessible to critical accounts such as an administrator account. Or, you might need to elevate the access permissions until the activities are complete. Both approaches can pose significant risks.

Critical accounts are those which can produce a business-critical outcome, whether cloud administrators or workload-specific privileged users. Compromise or misuse of such an account can have a detrimental-to-material effect on the business and its information systems. It's important to identify those accounts and adopt processes including close monitoring, and lifecycle management, including retirement.

Securing privileged access is a critical first step to establishing security assurances for business assets in a modern organization. The security of most or all business assets in an IT organization depends on the integrity of the privileged accounts used to administer, manage, and develop. Cyberattackers often target these accounts and other elements of privileged access to gain access to data, and systems using credential theft attacks like Pass-the-Hash, and Pass-the-Ticket.

Protecting privileged access against determined adversaries requires you to take a complete and thoughtful approach to isolate these systems from risks.

Are there any processes and tools leveraged to manage privileged activities?

Do not provide permanent access for any critical accounts and lower permissions when access is no longer required. Some strategies include:

- Just-in-time privileged access to Azure AD and Azure resources.
- Time-bound access.
- Approval-based access.
- Break glass for emergency access process to gain access.

Limit write access to production systems to service principals. No user accounts should have regular write-access.

Ensure there's a process for disabling or deleting administrative accounts that are unused.

You can use native and third-party options to elevate access permissions for at least highly privileged if not all activities. Azure AD Privileged Identity Management (Azure AD PIM) is the recommended native solution on Azure.

For more information about PIM, see What is Azure AD Privileged Identity Management?

Learn more

Establish lifecycle management for critical impact accounts

Related links

Back to the main article: Azure identity and access management considerations

Network security

11/19/2021 • 2 minutes to read • Edit Online

Protect assets by placing controls on network traffic originating in Azure, between on-premises and Azure hosted resources, and traffic to and from Azure. If security measures aren't in place attackers can gain access, for instance, by scanning across public IP ranges. Proper network security controls can provide defense-in-depth elements that help detect, contain, and stop attackers who gain entry into your cloud deployments.

Checklist

How have you secured the network of your workload?

- Segment your network footprint and create secure communication paths between segments. Align the network segmentation with overall enterprise segmentation strategy.
- Design security controls that identify and allow or deny traffic, access requests, and application communication between segments.
- Protect all public endpoints with Azure Front Door, Application Gateway, Azure Firewall, Azure DDoS Protection.
- Mitigate DDoS attacks with DDoS **Standard** protection for critical workloads.
- Prevent direct internet access of virtual machines.
- Control network traffic between subnets (east-west) and application tiers (north-south).
- Protect from data exfiltration attacks through a defense-in-depth approach with controls at each layer.

Azure security benchmark

The Azure Security Benchmark includes a collection of high-impact security recommendations you can use to help secure the services you use in Azure:

> The questions in this section are aligned to the Azure Security Benchmarks Network Security.

Azure services

- Azure Virtual Network
- Azure Firewall
- Azure ExpressRoute
- Azure Private Link

Reference architecture

Here are some reference architectures related to network security:

- Hub-spoke network topology in Azure
- Deploy highly available NVAs
- Windows N-tier application on Azure with SQL Server
- Azure Kubernetes Service (AKS) production baseline

Next steps

We recommend applying as many as of the best practices as early as possible, and then working to retrofit any gaps over time as you mature your security program.

Data protection

Related links

Combine network controls with application, identity, and other technical control types. This approach is effective in preventing, detecting, and responding to threats outside the networks you control. For more information, see these articles:

- Applications and services security
- Identity and access management considerations
- Data protection

Ensure that resource grouping and administrative privileges align to the segmentation model. For more information, see Administrative account security.

> Go back to the main article: Security

Implement network segmentation patterns on Azure

11/19/2021 • 8 minutes to read • Edit Online

A unified enterprise segmentation strategy guides technical teams to consistently segment access using networking, applications, identity, and any other access controls. Create segmentation in your network footprint by defining perimeters. The main reasons for segmentation are:

- The ability to group related assets that are a part of (or support) workload operations.
- Isolation of resources.
- Governance policies set by the organization.

Assume compromise is the recommended cybersecurity mindset and the ability to contain an attacker is vital in protecting information systems. Model an attacker able to achieve a foothold at various points within the workload and establish controls to mitigate further expansion.

Network controls can secure interactions between perimeters. This approach can strengthen the security posture and contain risks in a breach because the controls can detect, contain, and stop attackers from gaining access to an entire workload.

Containment of attack vectors within an environment is critical. However, to be effective in cloud environments, traditional approaches may prove inadequate and security organizations may need to evolve their methods.

Traditional segmentation approaches typically fail to achieve their goals as they have not been developed in a method to align with business use cases and application workloads. Often this results in overwhelming complexity requiring broad firewall exceptions.

An evolving emerging best practice recommendation is to adopt a Zero Trust strategy based on user, device, and application identities. In contrast to network access controls that are based on elements such as source and destination IP address, protocols, and port numbers, Zero Trust enforces and validates access control at *access time*. This avoids the need to play a prediction game for an entire deployment, network, or subnet — only the destination resource needs to provide the necessary access controls.

- Azure Network Security Groups can be used for basic layer 3 and 4 access controls between Azure Virtual Networks, their subnets, and the internet.
- Azure Web Application Firewall and the Azure Firewall can be used for more advanced network access controls that require application layer support.
- Local Admin Password Solution (LAPS) or a third-party Privileged Access Management can set strong local admin passwords and just-in-time access to them.

How does the organization implement network segmentation?

This article highlights some Azure networking features that create segments and restrict access to individual services.

> **IMPORTANT**
>
> Align your network segmentation strategy with the enterprise segmentation model. This will reduce confusion and challenges with different technical teams (networking, identity, applications, and so on). Each team should not develop their own segmentation and delegation models that don't align with each other.

Key points

- Create software-defined perimeters in your networking footprint and secure communication paths between them.
- Establish a complete zero trust segmentation strategy.
- Align technical teams in the enterprise on micro segmentation strategies for legacy applications.
- Azure Virtual Networks (VNets) are created in private address spaces. By default, no traffic is allowed between any two VNets. Open paths only when it's really needed.
- Use Network Security Groups (NSG) to secure communication between resources within a VNet.
- Use Application Security Groups (ASGs) to define traffic rules for the underlying VMs that run the workload.
- Use Azure Firewall to filter traffic flowing between cloud resources, the internet, and on-premise.
- Place resources in a single VNet, if you don't need to operate in multiple regions.
- If you need to be in multiple regions, have multiple VNets that are connected through peering.
- For advanced configurations, use a hub-spoke topology. A VNet is designated as a hub in a given region for all the other VNets as spokes in that region.

What is segmentation?

You can create software-defined perimeters in your networking footprint by using the various Azure services and features. When a workload (or parts of a given workload) is placed into separate segments, you can control traffic from/to those segments to secure communication paths. If a segment is compromised, you will be able to better contain the impact and prevent it from laterally spreading through the rest of your network. This strategy aligns with the key principle of Zero Trust model published by Microsoft that aims to bring world class security thinking to your organization.

Suggested actions

Create a risk containment strategy that blends proven approaches including:

- Existing network security controls and practices
- Native security controls available in Azure
- Zero trust approaches

Learn more

For information about creating a segmentation strategy, see Enterprise segmentation strategy.

Azure features for segmentation

When you operate on Azure, you have many segmentation options.

1. Subscription: A high-level construct, which provides platform powered separation between entities. It's intended to carve out boundaries between large organizations within a company and communication between resources in different subscriptions needs to be explicitly provisioned.

2. Virtual Network (VNets): Created within a subscription in private address spaces. They provide network level containment of resources with no traffic allowed by default between any two virtual networks. Like subscriptions, any communication between virtual networks needs to be explicitly provisioned.

3. Network Security Groups (NSG): An access control mechanisms for controlling traffic between resources within a virtual network and also with external networks, such as the internet, other virtual networks. NSGs can take your segmentation strategy to a granular level by creating perimeters for a subnet, a VM, or a group of VMs. For information about possible operations with subnets in Azure, see Subnets (Azure Virtual Networks).

4. Application Security Groups (ASGs): Similar to NSGs but are referenced with an application context. It allows you to group a set of VMs under an application tag and define traffic rules that are then applied to each of the underlying VMs.

5. Azure Firewall: A cloud native stateful Firewall as a service, which can be deployed in your VNet or in Azure Virtual WAN hub deployments for filtering traffic flowing between cloud resources, the internet, and on-premise. You create rules or policies (using Azure Firewall or Azure Firewall Manager) specifying allow/deny traffic using layer 3 to layer 7 controls. You can also filter traffic going to the internet using both Azure Firewall and third parties by directing some or all traffic through third-party security providers for advanced filtering & user protection.

Segmentation patterns

Here are some common patterns for segmenting a workload in Azure from a networking perspective. Each pattern provides a different type of isolation and connectivity. Choose a pattern based on your organization's needs.

Pattern 1: Single VNet

All the components of the workload reside in a single VNet. This pattern is appropriate you are operating in a single region because a VNet cannot span multiple regions.

Common ways for securing segments, such as subnets or application groups, are by using NSGs and ASGs. You can also use a Network Virtualized Appliance (NVAs) from Azure Marketplace or Azure Firewall to enforce and secure this segmentation.

In this image, Subnet1 has the database workload. Subnet2 has the web workloads. You can configure NSGs that allow Subnet1 to only communicate with Subnet2 and Subnet2 can only communicate with the internet.

Consider a use case where you have multiple workloads that are placed in separate subnets. You can place controls that will allow one workload to communicate to the backend of another workload.

Pattern 2: Multiple VNets that communicate through with peering

The resources are spread or replicated in multiple VNets. The VNets can communicate through peering. This pattern is appropriate when you need to group applications into separate VNets. Or, you need multiple Azure regions. One benefit is the built-in segmentation because you have to explicitly peer one VNet to another. Virtual network peering is not transitive. You can further segment within a VNet by using NSGs and ASGs as shown in pattern 1.

Pattern 3: Multiple VNets in a hub and spoke model

A VNet is designated as a *hub* in a given region for all the other VNets as *spokes* in that region. The hub and its spokes are connected through peering. All traffic passes through the hub that can act as a gateway to other hubs in different regions. In this pattern, the security controls are set up at the hubs so that they get to segment and govern the traffic in between other VNets in a scalable way. One benefit of this pattern is, as your network topology grows, the security posture overhead does not grow (except when you expand to new regions).

The recommended native option is Azure Firewall. This option works across both VNets and subscriptions to

govern traffic flows using layer 3 to layer 7 controls. You can define your communication rules and apply them consistently. Here are some examples:

- VNet 1 cannot communicate with VNet 2, but it can communicate VNet 3.
- VNet 1 cannot access public internet except for *.github.com.

With Azure Firewall Manager preview, you can centrally manage policies across multiple Azure Firewalls and enable DevOps teams to further customize local policies.

> **TIP**
>
> Here are some resources that illustrate provisioning of resources in a hub and spoke topology:
>
> GitHub: Hub and Spoke Topology Sandbox.
>
> The design considerations are described in Hub-spoke network topology in Azure.

Pattern comparison

CONSIDERATIONS	PATTERN 1	PATTERN 2	PATTERN 3
Connectivity/routing: how each segment communicates to each other	System routing provides default connectivity to any workload in any subnet.	Same as a pattern 1.	No default connectivity between spoke networks. A layer 3 router, such as the Azure Firewall, in the hub is required to enable connectivity.
Network level traffic filtering	Traffic is allowed by default. Use NSG, ASG to filter traffic.	Same as a pattern 1.	Traffic between spoke virtual networks is denied by default. Open selected paths to allow traffic through Azure Firewall configuration.
Centralized logging	NSG, ASG logs for the virtual network.	Aggregate NSG, ASG logs across all virtual networks.	Azure Firewall logs all accepted/denied traffic sent through the hub. View the logs in Azure Monitor.
Unintended open public endpoints	DevOps can accidentally open a public endpoint through incorrect NSG, ASG rules.	Same as a pattern 1.	Accidentally opened public endpoint in a spoke will not enable access because the return packet will get dropped through stateful firewall (asymmetric routing).
Application level protection	NSG or ASG provides network layer support only.	Same as a pattern 1.	Azure Firewall supports FQDN filtering for HTTP/S and MSSQL for outbound traffic and across virtual networks.

Next step

Secure network connectivity

Related links

- For information about setting up peering, see Virtual network peering.
- For best practices about using Azure Firewall in various configurations, see Azure Firewall Architecture Guide.
- For information about different access policies and control flow within a VNet, see Azure Virtual Network Subnet

Back to the main article: Network security

Azure services for securing network connectivity

11/19/2021 • 12 minutes to read • Edit Online

It's often the case that the workload and the supporting components of a cloud architecture will need to access external assets. These assets can be on-premises, devices outside the main virtual network, or other Azure resources. Those connections can be over the internet or networks within the organization.

Key points

- Protect non-public accessible services with network restrictions and IP firewall.
- Use Network Security Groups (NSGs) or Azure Firewall to protect and control traffic within the VNet.
- Use Service Endpoints or Private Link for accessing Azure PaaS services.
- Use Azure Firewall to protect against data exfiltration attacks.
- Restrict access to backend services to a minimal set of public IP addresses, only those services that really need it.
- Use Azure controls over third-party solutions for basic security needs. They're easy to configure and scale.
- Define access policies based on the type of workload and control flow between the different application tiers.

Connectivity between network segments

When designing a workload, you'll typically start by provisioning an Azure Virtual Network (VNet) in a private address space which has the workload. No traffic is allowed by default between any two virtual networks. If there's a need, define the communication paths explicitly. One way of connecting VNets is through Virtual network peering.

A key aspect is protecting the VMs in the VNet. The network interfaces on the VMs allow them to communicate with other VMs, the internet, and on-premises networks. To control traffic on VMs within a VNet (and subnet), use Application Security Groups (ASGs). You can group a set of VMs under an application tag and define traffic rules. Those rules are then applied to each of the underlying VMs.

A VNet is segmented into subnets based on business requirements. Provide proper network security controls that allow or deny inbound network traffic to, or outbound network traffic from, within larger network space.

You can also provision VMs with private IP addresses for protection. Take advantage of the Azure IP address to determine incoming traffic, how and where it's translated on to the virtual network.

A good Azure IP addressing schema provides flexibility, room for growth, and integration with on-premises networks. The schema ensures that communication works for deployed resources, minimizes public exposure of systems, and gives the organization flexibility in its network. If not properly designed, systems might not be able to communicate, and additional work will be required to remediate.

How do you isolate and protect traffic within the workload VNet?

To secure communication within a VNet, set rules that inspect traffic. Then, *allow* or *deny* traffic to, or from specific sources, and route them to the specified destinations.

> Review the rule set and confirm that the required services are not unintentionally blocked.

For traffic between subnets, the recommended way is through Network Security Groups (NSG). Define rules on each NSG that checks traffic to and from single IP address, multiple IP addresses, or entire subnets.

If NSGs are being used to isolate and protect the application, the rule set should be reviewed to confirm that required services are not unintentionally blocked, or more permissive access than expected is allowed. Azure Firewall (and Firewall Manager) can be used to centralize and manage firewall policies.

Another way is to use network virtual appliances (NVAs) that check inbound (ingress) and outbound (egress) traffic and filters based on rules.

How do you route network traffic through NVAs for security boundary policy enforcement, auditing, and inspection?

Use User Defined Routes (UDR) to control the next hop for traffic between Azure, on-premises, and internet resources. The routes can be applied to virtual appliance, virtual network gateway, virtual network, or internet.

For example, you need to inspect all ingress traffic from a public load balancer. One way is to host an NVA in a subnet that allows traffic only if certain criteria is met. That traffic is sent to the subnet that hosts an internal load balancer that routes that traffic to the backend services.

You can also use NVAs for egress traffic. For instance, all workload traffic is routed by using UDR to another subnet. That subnet has an internal load balancer that distributes requests to the NVA (or a set of NVAs). These NVAs direct traffic to the internet using their individual public IP addresses.

> **TIP**
>
> Here are the resources for the preceding example:
>
> GitHub: Automated failover for network virtual appliances.
>
> The design considerations are described in Deploy highly available NVAs.

Azure Firewall can serve as an NVA. Azure supports third-party network device providers. They're available in Azure Marketplace.

How do you get insights about ingoing and outgoing traffic of this workload?

As a general rule, configure and collect network traffic logs. If you use NSGs, capture and analyze NSG flow logs to monitor performance and security. The NSG flow logs enable Traffic Analytics to gain insights into internal and external traffic flows of the application.

For information about defining network perimeters, see Network segmentation.

Can the VNet and subnet handle growth?

Typically, you'll add more network resources as the design matures. Most organizations end up adding more resources to networks than initially planned. Refactoring to accommodate the extra resources is a labor-intensive process. There is limited security value in creating a very large number of small subnets and then trying to map network access controls (such as security groups) to each of them.

Plan your subnets based on roles and functions that use the same protocols. That way, you can add resources to the subnet without making changes to security groups that enforce network level access controls.

Don't use all open rules that allow inbound and outbound traffic to and from 0.0.0.0-255.255.255.255. Use a least-privilege approach and only allow relevant protocols. It will reduce your overall network attack surface on the subnet. All open rules provide a false sense of security because such a rule enforces no security.

The exception is when you want to use security groups only for network logging purposes.

Design virtual networks and subnets for growth. We recommend planning subnets based on common roles and

functions that use common protocols for those roles and functions. This allows you to add resources to the subnet without making changes to security groups that enforce network level access controls.

Suggested actions

Use NSG or consider using Azure Firewall to protect and control traffic within the VNET.

Learn more

- Azure firewall documentation
- Design virtual network subnet security
- Design an IP addressing schema for your Azure deployment
- Network security groups

Internet edge traffic

As you design the workload, consider security for internet traffic. Does the workload or parts of it need to be accessible from public IP addresses? What level of access should be given to prevent unauthorized access?

Internet edge traffic (also called *North-South traffic*) represents network connectivity between resources used by the workload and the internet. An internet edge strategy should be designed to mitigate as many attacks from the internet to detect or block threats. There are two primary choices that provide security controls and monitoring:

- Azure solutions such as Azure Firewall and Web Application Firewall (WAF).

Azure provides networking solutions to restrict access to individual services. Use multiple levels of security, such as combination of IP filtering, firewall rules to prevent application services from being accessed by unauthorized actors.

- Network virtual appliances (NVAs). You can use Azure Firewall or third-party solutions available in Azure Marketplace.

Azure security features are sufficient for common attacks, easy to configure, and scale. Third-party solutions often have advanced features but they can be hard to configure if they don't integrate well with fabric controllers. From a cost perspective, Azure options tend to be cheaper than partner solutions.

Information revealing the application platform, such as HTTP banners containing framework information (`X-Powered-By` , `X-ASPNET-VERSION`), are commonly used by malicious actors when mapping attack vectors of the application.

HTTP headers, error messages, and website footers should not contain information about the application platform. Azure CDN can be used to separate the hosting platform from end users. Azure API Management offers transformation policies that allow you to modify HTTP headers and remove sensitive information.

Suggested action

Consider using CDN for the workload to limit platform detail exposure to attackers.

Learn more

Azure CDN documentation

Communication with backend services

Most workloads are composed of multiple tiers where several services can serve each tier. Common examples of tiers are web front ends, business processes, reporting and analysis, backend infrastructure, and so on.

Application resources allowing multiple methods to publish app content (such as FTP, Web Deploy) should have the unused endpoints disabled. For Azure Web Apps, SCM is the recommended endpoint and it can be protected

separately with network restrictions for sensitive scenarios.

Public access to any workload should be judiciously approved and planned, as public entry points represent a key possible vector of compromise. When allowing access from public IPs to any back-end service, limiting the range of allowed IPs can significantly reduce the attack surface of that service. For example, if using Azure Front Door, you can limit backend tiers to allow Front Door IPs only; or if a partner uses your API, limit access to only their nominated public IP(s).

How do you configure traffic flow between multiple application tiers?

Use Azure Virtual Network Subnet to allocate separate address spaces for different elements or tiers within the workload. Then, define different access policies to control traffic flows between those tiers and restrict access. You can implement those restrictions through IP filtering or firewall rules.

Do you need to restrict access to the backend infrastructure?

Restrict access to backend services to a minimal set of public IP addresses with App Services IP restrictions or Azure Front Door.

Web applications typically have one public entry point and don't expose subsequent APIs and database servers over the internet. Expose only a minimal set of public IP addresses based on need *and* only those who really need it. For example, when using gateway services, such as Azure Front Door, it's possible to restrict access only to a set of Front Door IP addresses and lock down the infrastructure completely.

Suggested action

- Restrict and protect application publishing methods.

Learn more

- Set up Azure App Service access restrictions
- Azure Front Door documentation
- Deploy your app to Azure App Service using FTP/S

Connection with Azure PaaS services

The workload will often need to communicate with other Azure services. For example, it might need to get secrets from Azure Key Vault. Avoid making connections over the public internet.

Does the workload use secure ways to access Azure PaaS services?

Common approaches for accessing PaaS services are Service Endpoints or Private Links. Both approaches restrict access to PaaS endpoints only from authorized virtual networks, effectively mitigating data intrusion risks and associated impact to application availability.

With Service Endpoints, the communication path is secure because you can reach the PaaS endpoint without needing a public IP address on the VNet. Most PaaS services support communication through service endpoints. For a list of generally available services, see Virtual Network service endpoints.

Another mechanism is through Azure Private Link. Private Endpoint uses a private IP address from your VNet, effectively bringing the service into your VNet. For details, see What is Azure Private Link?.

Service Endpoints provide service level access to a PaaS service, whereas Private Link provides direct access to a specific PaaS resource to mitigate data exfiltration risks, such as malicious admin access. Private Link is a paid service and has meters for inbound and outbound data processed. Private Endpoints are also charged.

How do you control outgoing traffic of Azure PaaS services where Private Link isn't available?

Use NVAs and Azure Firewall (for supported protocols) as a reverse proxy to restrict access to only authorized PaaS services for services where Private Link isn't supported. Use Azure Firewall to protect against data exfiltration concerns.

On-premises to cloud connectivity

In a hybrid architecture, the workload runs partly on-premises and partly in Azure. Have security controls that check traffic entering Azure virtual network from on-premises data center.

How do you establish cross premises connectivity?

Use Azure ExpressRoute to set up cross premises connectivity to on-premises networks. This service uses a private, dedicated connection through a third-party connectivity provider. The private connection extends your on-premises network into Azure. This way, you can reduce the risk of potential of access to company's information assets on-premises.

How do you access VMs?

Use Azure Bastion to log into your VMs and avoid public internet exposure using SSH and RDP with private IP addresses only. You can also disable RDP/SSH access to VMs and use VPN, ExpressRoute to access these virtual machines for remote management.

Do the cloud or on-premises VMs have direct internet connectivity for users that may perform interactive logins?

Attackers constantly scan public cloud IP ranges for open management ports and attempt low-cost attacks such as common passwords and known unpatched vulnerabilities. Develop processes and procedures to prevent direct internet access of VMs with logging and monitoring to enforce policies.

How is internet traffic routed?

Decide how to route internet traffic. You can use on-premises security devices (also called *forced tunneling*) or allow connectivity through cloud-based network security devices.

For production enterprise, allow cloud resources to start and respond to internet request directly through cloud network security devices defined by your internet edge strategy. This approach fits the Nth datacenter paradigm, that is Azure datacenters are a part of your enterprise. It scales better for an enterprise deployment because it removes hops that add load, latency, and cost.

Another option is to force tunnel all outbound internet traffic from on-premises through site-to-site VPN. Or, use a cross-premise WAN link. Network security teams have greater security and visibility to internet traffic. Even when your resources in the cloud try to respond to incoming requests from the internet, the responses are force tunneled. This option fits a datacenter expansion use case and can work well for a quick proof of concept, but scales poorly because of the increased traffic load, latency, and cost. For those reasons, we recommend that you avoid forced tunneling.

Next step

Secure endpoints

Related links

For information about controlling next hop for traffic, see Azure Virtual Network User Defined Routes (UDR).

For information about web application firewalls, see Application Gateway WAF.

For information about Network Appliances from Azure Marketplace, see Network Appliances.

For information about cross premises connectivity, see Azure site-to-site VPN or ExpressRoute.

For information about using VPN/ExpressRoute to access these virtual machines for remote management, see Disable RDP/SSH access to Azure Virtual Machines.

Go back to the main article: Network security

Best practices for endpoint security on Azure

11/19/2021 • 7 minutes to read • Edit Online

An *endpoint* is an address exposed by a web application so that external entities can communicate with it. A malicious or an inadvertent interaction with the endpoint can compromise the security of the application and even the entire system. One way to protect the endpoint is by placing filter controls on the network traffic that it receives, such as defining rule sets. A defense-in-depth approach can further mitigate risks. Include supplemental controls that protect the endpoint if the primary traffic controls fail.

This article describes way in which you can protect web applications with Azure services and features. For product documentation, see Related links.

Key points

- Protect all public endpoints with Azure Front Door, Application Gateway, Azure Firewall, Azure DDoS Protection.
- Use web application firewall (WAF) to protect web workloads.
- Protect workload publishing methods and restrict ways that are not in use.
- Mitigate DDoS attacks. Use Standard protection for critical workloads where outage would have business impact. Also consider CDN as another layer of protection.
- Develop processes and procedures to prevent direct internet access of virtual machines (such as proxy or firewall) with logging and monitoring to enforce policies.
- Implement an automated and gated CI/CD deployment process.

Public endpoints

A public endpoint receives traffic over the internet. The endpoints make the service easily accessible to attackers.

Service Endpoints and Private Link can be leveraged to restrict access to PaaS endpoints only from authorized virtual networks, effectively mitigating data intrusion risks and associated impact to application availability. Service Endpoints provide service level access to a PaaS service, while Private Link provides direct access to a specific PaaS resource to mitigate data exfiltration risks such as malicious admin scenarios.

Configure service endpoints and private links where appropriate.

Are all public endpoints of this workload protected?

An initial design decision is to assess whether you need a public endpoint at all. If you do, protect it by using these mechanisms.

For more information, see Virtual Network service endpoints and What is Azure Private Endpoint?

Web application firewalls (WAFs)

WAFs provide a basic level of security for web applications. WAFs are appropriate if the organizations that have invested in application security as WAFs provide additional defense-in-depth mitigation.

WAFs mitigate the risk of an attacker to exploit commonly seen security vulnerabilities for applications. WAFs provide a basic level of security for web applications. This mechanism is an important mitigation because attackers target web applications for an ingress point into an organization (similar to a client endpoint).

External application endpoints should be protected against common attack vectors, from Denial of Service (DoS) attacks like Slowloris to app-level exploits, to prevent potential application downtime due to malicious intent.

Azure-native technologies such as Azure Firewall, Application Gateway/Azure Front Door, WAF, and DDoS Protection Standard Plan can be used to achieve requisite protection (Azure DDoS Protection).

Azure Application Gateway has WAF capabilities to inspect web traffic and detect attacks at the HTTP layer. It's a load balancer and HTTP(S) full reverse proxy that can do secure socket layer (SSL) encryption and decryption.

For example, your workload is hosted in Application Service Environments(ILB ASE). The APIs are consolidated internally and exposed to external users. This external exposure could be achieved using an Application Gateway. This service is a load balancer. It forwards request to the internal API Management service, which in turn consumes the APIs deployed in the ASE. Application Gateway is also configured over port 443 for secured and reliable outbound calls.

> **TIP**
>
> The design considerations for the preceding example are described in Publishing internal APIs to external users.

Azure Front Door and Azure Content Delivery Network (CDN) also have WAF capabilities.

Suggestion actions

Protect all public endpoints with appropriate solutions such as Azure Front Door, Application Gateway, Azure Firewall, Azure DDOS Protection, or any third-party solution.

Learn more

- What is Azure Firewall?
- Azure DDoS Protection Standard overview
- Azure Front Door documentation
- What is Azure Application Gateway?

Azure Firewall

Protect the entire virtual network against potentially malicious traffic from the internet and other external locations. It inspects incoming traffic and only passes the allowed requests to pass through.

A common design is to implement a DMZ or a perimeter network in front of the application. The DMZ is a separate subnet with the firewall.

> **TIP**
>
> The design considerations are described in Deploy highly available NVAs.

Combination approach

When you want higher security and there's a mix of web and non-web workloads in the virtual network use both Azure Firewall and Application Gateway. There are several ways in which those two services can work together.

For example, you want to filter egress traffic. You want to allow connectivity to a specific Azure Storage Account but not others. You'll need fully qualified domain name (FQDN)-based filters. In this case run Firewall and Application Gateway in parallel.

Another popular design is when you want Azure Firewall to inspect all traffic and WAF to protect web traffic, and the application needs to know the client's source IP address. In this case, place Application Gateway in front of Firewall. Conversely, you can place Firewall in front of WAF if you want to inspect and filter traffic before it reaches the Application Gateway.

For more information, see Firewall and Application Gateway for virtual networks.

It's challenging to write concise firewall rules for networks where different cloud resources dynamically spin up and down. Use Microsoft Defender for Cloud to detect misconfiguration risks.

Authentication

Disable insecure legacy protocols for internet-facing services. Legacy authentication methods are among the top attack vectors for cloud-hosted services. Those methods don't support other factors beyond passwords and are prime targets for password spraying, dictionary, or brute force attacks.

Mitigate DDoS attacks

In a distributed denial-of-service (DDoS) attack, the server is overloaded with fake traffic. DDoS attacks are common and can be debilitating. An attack can completely block access or take down services. Make sure all business-critical web application and services have DDoS mitigation beyond the default defenses so that the application doesn't experience downtime because that can negatively impact business.

Microsoft recommends adopting advanced protection for any services where downtime will have negative impact on the business.

How do you implement DDoS protection?

Here are some considerations:

- DDoS protection at the infrastructure level in which your workload runs. Azure infrastructure has built-in defenses for DDoS attacks.
- DDoS protection at the network (layer 3) layer. Azure provides additional protection for services provisioned in a virtual network.
- DDoS protection with caching. Content delivery network (CDN) can add another layer of protection. In a DDoS attack, a CDN intercepts the traffic and stops it from reaching the backend server. Azure CDN is natively protected. Azure also supports popular CDNs that are protected with proprietary DDoS mitigation platform.
- Advanced DDoS protection. In your security baseline, consider features with monitoring techniques that use machine learning to detect anomalous traffic and proactively protect your application before service degradation occurs.

For example, the Windows N-tier application on Azure with SQL Server reference architecture uses Azure DDoS Protection Standard because this option:

- Uses adaptive tuning, based on the application's network traffic patterns, to detect threats.
- Guarantees 100% SLA.
- Can be cost effective. For example, during a DDoS attack, the first set of attacks cause the provisioned resources to scale out. For a resource such as a virtual machine scale set, 10 machines can grow to 100, increasing overall costs. With Standard protection, you don't have to worry about the cost of the scaled resources because Azure will provide a cost credit.

For information about Azure DDoS Protection services, see Azure DDoS Protection Standard documentation.

Suggested action

Identify critical workloads that are susceptible to DDoS attacks and enable Distributed Denial of Service (DDoS) mitigations for all business-critical web applications and services.

Learn more

For a list of reference architectures that demonstrate the use of DDoS protection, see Azure DDoS Protection reference architectures.

Adopt DevOps

Developers shouldn't publish their code directly to app servers.

Does the organization have an CI/CD process for publishing code in this workload?

Implement lifecycle of continuous integration, continuous delivery (CI/CD) for applications. Have processes and tools in place that aid in an automated and gated CI/CD deployment process.

How are the publishing methods secured?

Application resources allowing multiple methods to publish app content, such as FTP, Web Deploy should have the unused endpoints disabled. For Azure Web Apps, SCM is the recommended endpoint. It can be protected separately with network restrictions for sensitive use cases.

Next step

Data flow

Related links

- Azure Firewall
- What is Azure Web Application Firewall on Azure Application Gateway?
- Azure DDoS Protection Standard

Go back to the main article: Network security

Traffic flow security in Azure

11/19/2021 • 3 minutes to read • Edit Online

Protect data anywhere it goes including cloud services, mobile devices, workstations, or collaboration platforms. In addition to using access control and encryption mechanisms, apply strong network controls that detect, monitor, and contain attacks.

Key points

- Control network traffic between subnets (east-west) and application tiers (north-south).
- Apply a layered defense-in-depth approach that starts with Zero-Trust policies.
- Use a cloud application security broker (CASB).

East-west and north-south traffic

When analyzing the network flow of a workload, distinguish between east-west traffic from north-south traffic. Most cloud architectures use a combination of both types.

Is the traffic between subnets, Azure components and tiers of the workload managed and secured?

- **North-south traffic**

  North-south refers to the traffic that flows in and out of a datacenter. For example, traffic from an application to a backend service. This type of traffic is a typical target for attack vectors because it flows over the public internet. Proper network controls must be in place so that the queries to and from a data center are secure.

  Consider a typical flow in an Azure Kubernetes Service (AKS) cluster. The cluster receives incoming (ingress) traffic from HTTP requests. The cluster can also send outgoing (egress) traffic to send queries to other services, such as pulling a container image.

  Your design can use Web Application Firewall on Application Gateway to secure ingress traffic, and Azure Firewall to secure outgoing (egress) traffic.

- **East-west traffic**

  East-west traffic refers to traffic between or within data centers. For this type of traffic, several resources of the network infrastructure communicate with each other. Those resources can be virtual networks, subnets within those virtual networks, and so on. Security of east-west traffic can get overlooked even though it makes up a large portion of the workload traffic. It's assumed that the infrastructure firewalls are sufficient to block attacks. Make sure there are proper controls between network resources.

  Extending the example of the AKS cluster to this concept, east-west traffic is the traffic within the cluster. For example, communication between pods, such as the ingress controller and the workload. If your workload is composed of multiple applications, the communication between those applications would fall into this category.

  By using Kubernetes network policies, you can restrict which pods can communicate, starting from a Zero-Trust policy and then opening specific communication paths as needed.

> **TIP**
>
> Here are the resources for the preceding AKS example:
>
> GitHub: Azure Kubernetes Service (AKS) Secure Baseline Reference Implementation.
>
> The design considerations are described in Azure Kubernetes Service (AKS) production baseline.

Data exfiltration

Data exfiltration is a common attack where an internal or external malicious actor does an unauthorized data transfer. Most often access is gained because of lack of network controls.

Network virtual appliance (NVA) solutions and Azure Firewall (for supported protocols) can be leveraged as a reverse proxy to restrict access to only authorized PaaS services for services where Private Link is not yet supported (Azure Firewall).

Configure Azure Firewall or a third-party next generation firewall to protect against data exfiltration concerns.

Are there controls in the workload design to detect and protect from data exfiltration?

Choose a defense-in-depth design that can protect network communications at various layers, such as a hub-spoke topology. Azure provides several controls to support the layered design:

- Use Azure Firewall to allow or deny traffic using layer 3 to layer 7 controls.
- Use Azure Virtual Network User Defined Routes (UDR) to control next hop for traffic.
- Control traffic with Network Security Groups (NSGs) between resources within a virtual network, internet, and other virtual networks.
- Secure the endpoints through Azure PrivateLink and Private Endpoints.
- Detect and protect at deep levels through packet inspection.
- Detect attacks and respond to alerts through Microsoft Sentinel and Microsoft Defender for Cloud.

> **IMPORTANT**
>
> Network controls are not sufficient in blocking data exfiltration attempts. Harden the protection with proper identity controls, key protection, and encryption. For more information, see these sections:
>
> - Data protection considerations
> - Identity and access management considerations

Have you considered a cloud application security broker (CASB) for this workload?

CASBs provide a central point of control for enforcing policies. They provide rich visibility, control over data travel, and sophisticated analytics to identify and combat cyberthreats across all Microsoft and third-party cloud services.

Learn more

- Azure firewall documentation
- Azure Marketplace networking apps

Related links

- Azure Firewall
- Network Security Groups (NSG)
- What is Azure Web Application Firewall on Azure Application Gateway?
- What is Azure Private Link?

Go back to the main article: Network security

Data protection considerations

11/19/2021 • 2 minutes to read • Edit Online

Classify, protect, and monitor sensitive data assets using access control, encryption, and logging in Azure. Provide controls on data at rest and in transit.

Checklist

How are you managing encryption for this workload?

- Use identity based storage access controls.
- Use built-in features for data encryption for Azure services.
- Classify all stored data and encrypt it.
- Protect data moving over a network through encryption at all points so that it's not accessed unauthorized users.
- Store keys in managed key vault service with identity-based access control and audit policies.
- Rotate keys and other secrets frequently.

Azure security benchmark

The Azure Security Benchmark includes a collection of high-impact security recommendations you can use to help secure the services you use in Azure:

> The questions in this section are aligned to the Azure Security Benchmarks Data Protection.

Reference architecture

Here are some reference architectures related to secure storage:

- Using Azure file shares in a hybrid environment
- DevSecOps in Azure

Next steps

We recommend that you review the practices and tools implemented as part of the development cycle.

Data protection

Related links

> Back to the main article: Security

Data encryption in Azure

11/19/2021 • 9 minutes to read • Edit Online

Data can be categorized by its state:

- **Data at rest**. All information storage objects, containers, and types that exist statically on physical media, whether magnetic or optical disk.
- **Data in transit**. Data that is being transferred between components, locations, or programs.

In a cloud solution, a single business transaction can lead to multiple data operations where data moves from one storage medium to another. To provide complete data protection, it must be encrypted on storage volumes and while it's transferred from one point to another.

Key points

- Use identity-based storage access controls.
- Use standard and recommended encryption algorithms.
- Use only secure hash algorithms (SHA-2 family).
- Classify your data at rest and use encryption.
- Encrypt virtual disks.
- Use an additional key encryption key (KEK) to protect your data encryption key (DEK).
- Protect data in transit through encrypted network channels (TLS/HTTPS) for all client/server communication. Use TLS 1.2 on Azure.

Azure encryption features

Azure provides built-in features for data encryption in many layers that participate in data processing. We recommend that for each service, enable the encryption capability. The encryption is handled automatically using Azure-managed keys. This almost requires no user interaction.

We recommend implementing identity-based storage access controls. Authentication with a shared key (like a Shared Access Signature) doesn't permit the same flexibility and control as identity-based access control. The leak of a shared key might allow indefinite access to a resource, whereas a role-based access control can be identified and authenticated more strongly.

Storage in a cloud service like Azure is architected and implemented quite differently than on-premises solutions to enable massive scaling, modern access through REST APIs, and isolation between tenants. Cloud service providers make multiple methods of access control over storage resources available. Examples include shared keys, shared signatures, anonymous access, and identity provider-based methods.

Consider some built-in features of Azure Storage:

- **Identity-based access**. Supports access through Azure Active Directory (Azure AD) and key-based authentication mechanisms, such as Symmetric Shared Key Authentication, or Shared Access Signature (SAS).
- **Built-in encryption**. All stored data is encrypted by Azure storage. Data cannot be read by a tenant if it has not been written by that tenant. This feature provides control over cross tenant data leakage.
- **Region-based controls**. Data remains only in the selected region and three synchronous copies of data are maintained within that region. Azure storage provides detailed activity logging is available on an opt-in basis.
- **Firewall features**. The firewall provides an additional layer of access control and storage threat protection

to detect anomalous access and activities.

For the complete set of features, see Azure Storage Service encryption.

Suggested action

Identify provider methods of authentication and authorization that are the least likely to be compromised, and enable more fine-grained role-based access controls over storage resources.

Learn more

For more information, reference Authorize access to blobs using Azure Active Directory.

Standard encryption algorithms

Does the organization use industry standard encryption algorithms instead of creating their own?

Organizations should not develop and maintain their own encryption algorithms. Avoid using custom encryption algorithms or direct cryptography in your workload. These methods rarely stand up to real world attacks.

Secure standards already exist on the market and should be preferred. If custom implementation is required, developers should use well-established cryptographic algorithms and secure standards. Use Advanced Encryption Standard (AES) as a symmetric block cipher, AES-128, AES-192, and AES-256 are acceptable.

Developers should use cryptography APIs built into operating systems instead of non-platform cryptography libraries. For .NET, follow the .NET Cryptography Model.

We advise using standard and recommended encryption algorithms.

For more information, refer to Choose an algorithm.

Are modern hashing functions used?

Applications should use the SHA-2 family of hash algorithms (SHA-256, SHA-384, SHA-512).

Data at rest

All important data should be classified and encrypted with an encryption standard. Classify and protect all information storage objects. Use encryption to make sure the contents of files cannot be accessed by unauthorized users.

Data at rest is encrypted by default in Azure, but is your critical data classified and tagged, or labeled so that it can be audited?

Your most sensitive data might include business, financial, healthcare, or personal information. Discovering and classifying this data can play a pivotal role in your organization's information protection approach. It can serve as infrastructure for:

- Helping to meet standards for data privacy and requirements for regulatory compliance.
- Various security scenarios, such as monitoring (auditing) and alerting on anomalous access to sensitive data.
- Controlling access to and hardening the security of databases that contain highly sensitive data.

Suggested action

Classify your data. Consider using Data Discovery & Classification in Azure SQL Database.

Data classification

A crucial initial exercise for protecting data is to organize it into categories based on certain criteria. The classification criteria can be your business needs, compliance requirements, and the type of data.

Depending on the category, you can protect it through:

- Standard encryption mechanisms.
- Enforce security governance through policies.
- Conduct audits to make sure the security measures are compliant.

One way of classifying data is through the use of tags.

Does the organization encrypt virtual disk files for virtual machines that are associated with this workload?

There are many options to store files in the cloud. Cloud-native apps typically use Azure Storage. Apps that run on VMs use them to store files. VMs use virtual disk files as virtual storage volumes and exist in a blob storage.

Consider a hybrid solution. Files can move from on-premises to the cloud, from the cloud to on-premises, or between services hosted in the cloud. One strategy is to make sure that the files and their contents aren't accessible to unauthorized users. You can use authentication-based access controls to prevent unauthorized downloading of files. However, that is not enough. Have a backup mechanism to secure the virtual disk files in case authentication and authorization or its configuration is compromised. There are several approaches. You can encrypt the virtual disk files. If an attempt is made to mount disk files, the contents of the files cannot be accessed because of the encryption.

We recommend that you enable virtual disk encryption. For information about how to encrypt Windows VM disks, see Quickstart: Create and encrypt a Windows VM with the Azure CLI.

Azure-based virtual disks are stored as files in a Storage account. If no encryption is applied to a virtual disk, and an attacker manages to download a virtual disk image file, it can be mounted and inspected at the attacker's leisure as if they had physical access to the source computer. Encrypting virtual disk files helps prevent attackers from gaining access to the contents of those disk files in the event they are able to download them. Depending on the sensitivity of the information stored on the disk, unencrypted access could represent a critical risk to confidential business data (such as a SQL database) or identity (such as an AD Domain Controller).

An example of virtual disk encryption is Azure Disk Encryption.

Azure Disk Encryption helps protect and safeguard your data to meet your organizational security and compliance commitments. It uses the Bitlocker-feature of Windows (or DM-Crypt on Linux) to provide volume encryption for the OS and data disks of Azure virtual machines (VMs). It is integrated with Azure Key Vault to help you control and manage the disk encryption keys, and secrets.

Virtual machines use virtual disk files as storage volumes and exist in a cloud service provider's blob storage system. These files can be moved from on-premises to cloud systems, from cloud systems to on-premises, or between cloud systems. Due to the mobility of these files, it's recommended that the files and the contents are not accessible to unauthorized users.

Does the organization use identity-based storage access controls for this workload?

There are many ways to control access to data: shared keys, shared signatures, anonymous access, identity provider-based. Use Azure Active Directory (Azure AD) and role-based access control (RBAC) to grant access. For more information, see Identity and access management considerations.

Does the organization protect keys in this workload with an additional key encryption key (KEK)?

Use more than one encryption key in an encryption at rest implementation. Storing an encryption key in Azure Key Vault ensures secure key access and central management of keys.

Use an additional key encryption key (KEK) to protect your data encryption key (DEK).

Suggested actions

Identify unencrypted virtual machines via Microsoft Defender for Cloud or script, and encrypt via Azure Disk Encryption. Ensure all new virtual machines are encrypted by default and regularly monitor for unprotected disks.

Learn more

Azure Disk Encryption for virtual machines and virtual machine scale sets

Data in transit

Data in transit should be encrypted at all points to ensure data integrity.

Protecting data in transit should be an essential part of your data protection strategy. Because data is moving back and forth from many locations, we generally recommend that you always use SSL/TLS protocols to exchange data across different locations.

For data moving between your on-premises infrastructure and Azure, consider appropriate safeguards such as HTTPS or VPN. When sending encrypted traffic between an Azure virtual network and an on-premises location over the public internet, use Azure VPN Gateway.

Does the workload communicate over encrypted network traffic only?

Any network communication between client and server where man-in-the-middle attacks can occur, must be encrypted. All website communication should use HTTPS, no matter the perceived sensitivity of transferred data. Man-in-the-middle attacks can occur anywhere on the site, not just login forms.

This mechanism can be applied to use cases such as:

- Web applications and APIs for all communication with clients.
- Data moving across a service bus from on-premises to the cloud and other way around, or during an input/output process.

In certain architecture styles such as microservices, data must be encrypted during communication between the services.

What TLS version is used across workloads?

Using the latest version of TLS is preferred. All Azure services support TLS 1.2 on public HTTPS endpoints. Migrate solutions to support TLS 1.2 and use this version by default.

When traffic from clients using older versions of TLS is minimal, or it's acceptable to fail requests made with an older version of TLS, consider enforcing a minimum TLS version. For information about TLS support in Azure Storage, see Remediate security risks with a minimum version of TLS.

Sometimes you need to isolate your entire communication channel between your on-premises and the cloud infrastructure by using either a virtual private network (VPN) or ExpressRoute. For more information, see these articles:

- Extending on-premises data solutions to the cloud
- Configure a Point-to-Site VPN connection to a VNet using native Azure certificate authentication: Azure portal

For more information, see Protect data in transit.

Is there any portion of the application that doesn't secure data in transit?

All data should be encrypted in transit using a common encryption standard. Determine if all components in the solution are using a consistent standard. There are times when encryption is not possible because of technical limitations, make sure the reason is clear and valid.

Suggested actions

Identify workloads using unencrypted sessions and configure the service to require encryption.

Learn more

- Encrypt data in transit
- Azure encryption overview

Next steps

While it's important to protect data through encryption, it's equally important to protect they keys that provide access to the data.

Key and secret management

Related links

Identity and access management services authenticate and grant permission to users, partners, customers, applications, services, and other entities. For security considerations, see Azure identity and access management considerations.

Back to the main article: Data protection

Key and secret management considerations in Azure

11/19/2021 • 5 minutes to read • Edit Online

Encryption is an essential tool for security because it restricts access. However, it's equally important to protect the secrets (keys, certificates) key that provide access to the data.

Key points

- Use identity-based access control instead of cryptographic keys.
- Use standard and recommended encryption algorithms.
- Store keys and secrets in managed key vault service. Control permissions with an access model.
- Rotate keys and other secrets frequently. Replace expired or compromised secrets.

Identity-based access control

Organizations shouldn't develop and maintain their own encryption algorithms. There are many ways to provide access control over storage resources available, such as:

- Shared keys
- Shared signatures
- Anonymous access
- Identity provider-based methods

Secure standards already exist on the market and should be preferred. AES should be used as symmetric block cipher, `AES-128`, `AES-192`, and `AES-256` are acceptable. Crypto APIs built into operating systems should be used where possible, instead of non-platform crypto libraries. For .NET, make sure you follow the .NET Cryptography Model.

Do you prioritize authentication through identity services for a workload over cryptographic keys?

Protection of cryptographic keys can often get overlooked or implemented poorly. Managing keys securely with application code is especially difficult and can lead to mistakes such as accidentally publishing sensitive access keys to public code repositories.

Use of identity-based options for storage access control is recommended. This option uses role-based access controls (RBAC) over storage resources. Use RBAC to assign permissions to users, groups, and applications at a certain scope. Identity systems such as Azure Active Directory (Azure AD) offer secure and usable experience for access control with built-in mechanisms for handling key rotation, monitoring for anomalies, and others.

> **NOTE**
> Grant access based on the principle of least privilege. Risk of giving more privileges than necessary can lead to data compromise.

Suppose you need to store sensitive data in Azure Blob Storage. You can use Azure AD and RBAC to authenticate a service principal that has the required permissions to access the storage. For more information about the feature, reference Authorize access to blobs and queues using Azure Active Directory.

> **TIP**
> Using SAS tokens is a common way to control access. SAS tokens are created by using the service owner's Azure AD credentials. The tokens are created per resource and you can use Azure RBAC to restrict access. SAS tokens have a time limit, which controls the window of exposure. Here are the resources for the preceding example:
>
> GitHub: Azure Cognitive Services Reference Implementation.
>
> The design considerations are described in Speech transcription with Azure Cognitive Services.

Key storage

To prevent security leaks, store the following keys and secrets in a secure store:

- API keys
- Database connection strings
- Data encryption keys
- Passwords

Sensitive information shouldn't be stored within the application code or configuration. An attacker gaining read access to source code shouldn't gain knowledge of application and environment-specific secrets.

Store all application keys and secrets in a managed key vault service such as Azure Key Vault or HashiCorp Vault. Storing encryption keys in a managed store further limits access. The workload can access the secrets by authenticating against Key Vault by using managed identities. That access can be restricted with Azure RBAC.

Make sure no keys and secrets for any environment types (Dev, Test, or Production) are stored in application configuration files or CI/CD pipelines. Developers can use Visual Studio Connected Services or local-only files to access credentials.

Have processes that periodically detect exposed keys in your application code. An option is Credential Scanner. For information about the configuring task, reference Credential Scanner task.

Do you have an access model for key vaults to grant access to keys and secrets?

To secure access to your key vaults, control permissions to keys and secrets through an access model. For more information, reference Access model overview.

Suggested actions

Consider using Azure Key Vault for secrets and keys.

Operational considerations

Who is responsible for managing keys and secrets in the application context?

Key and certificate rotation is often the cause of application outages. Even Azure has experienced expired certificates. It's critical that the rotation of keys and certificates be scheduled and fully operationalized. The rotation process should be automated and tested to ensure effectiveness. Azure Key Vault supports key rotation and auditing.

Central SecOps team provides guidance on how keys and secrets are managed (governance). Application DevOps team is responsible for managing the application-related keys and secrets.

What types of keys and secrets are used and how are those generated?

The following approaches include:

- Microsoft-managed Keys
- Customer-managed Keys
- Bring Your Own Key

The decision is often driven by security, compliance, and specific data classification requirements. Develop a clear understanding of these requirements to determine the most suitable type of keys.

Are keys and secrets rotated frequently?

To reduce the attack vectors, secrets require rotation and are prone to expiration. The process should be automated and executed without any human interactions. Storing them in a managed store simplifies those operational tasks by handling key rotation.

Replace secrets after they've reached the end of their active lifetime or if they've been compromised. Renewed certificates should also use a new key. Have a process for situations where keys get compromised (leaked) and need to be regenerated on-demand. For example, secrets rotation in SQL Database.

For more information, reference Key Vault Key Rotation.

By using managed identities, you remove the operational overhead for storing the secrets or certificates of service principals.

Are the expiration dates of SSL/TLS certificates monitored and are processes in place to renew them?

A common cause of application outage is expired SSL/TLS certificates.

Avoid outages by tracking the expiration dates of SSL/TLS certificates and renewing them in due time. Ideally, the process should be automated, although this often depends on used certificate authority (CA). If not automated, use alerts to make sure expiration dates don't go unnoticed.

Suggested actions

Implement a process for SSL certificate management and the automated renewal process with Azure Key Vault.

Learn more

Tutorial: Configure certificate auto-rotation in Key Vault

Related content

Identity and access management services authenticate and grant permission to the following groups:

- Users
- Partners
- Customers
- Applications
- Services
- Other entities

For security considerations, reference Azure identity and access management considerations.

Back to the main article: Data protection

Next steps

Protect data at rest and in transit through encryption. Make sure you use standard encryption algorithms.

Data encryption

Applications and services

11/19/2021 • 2 minutes to read • Edit Online

Applications and the data associated with them act as the primary store of business value on a cloud platform. Applications can play a role in risks to the business because:

- **Business processes** are encapsulated and executed by applications and services need to be available and provided with high integrity.
- **Business data** is stored and processed by application workloads and requires high assurances of confidentiality, integrity, and availability.

Identify and classify business critical applications

Enterprise organizations typically have a large application portfolio, but not all applications have equal importance. Applications containing business-critical data, regulated data, and with high business value, visibility, or criticality should be identified and classified, to direct investment of monitoring, time, and resources appropriately. You should also identify applications or systems with significant access — those which might grant control over other critical systems or data.

Suggested actions

Identify and classify key organizational applications according to organizational impact.

Next steps

See these best practices related to PaaS applications.

Securing PaaS deployments

Secure communication paths between applications and the services. Make sure that there's a distinction between the endpoints exposed to the public internet and private ones. Also, the public endpoints are protected with web application firewall.

Network security

Application classification for security

11/19/2021 • 5 minutes to read • Edit Online

Azure can host both legacy and modern applications through Infrastructure as a Service (IaaS) virtual machines and Platform as a Service (PaaS). With legacy applications, you have the responsibility of securing all dependencies including OS, middleware, and other components. For PaaS applications, you don't need to manage and secure the underlying server OS. You are responsible for the application configuration.

This article describes the considerations for understanding the hosting models and the security responsibility of each, identifying critical applications.

Understand your responsibility as an owner

Securing an application requires security assurances for three aspects:

- **Application code**. The logic that defines the custom application that you write. Securing that code requires identifying and mitigating risks from the design and implementation of the application and assessing supply chain risk of included components.
- **Application services**. The cloud services that the application uses such as databases, identity providers, event hubs, IoT device management, and so on. Security for cloud services is a shared responsibility. The cloud provider ensures the security of the underlying service. The application owner is responsible for security implications of the configuration and operation of the service instance(s) used by the application including any data stored and processed on the service.
- **Application hosting platform**. The computing environment where the application runs. This could take many forms with significant variations on who is responsible for security:
  - **Legacy applications**. typically require a full operating system (and any middleware) hosted on physical or virtualized hardware. This operating system and installed middleware/other components are operated and secured by the application owner or their infrastructure team(s). The security responsibility for the physical hardware and OS virtualization components (virtualization hosts, operating systems, and management services) varies:

    - **On-premises:** The application owner is responsible for maintenance and security.
    - **IaaS:** The cloud provider is responsible for the underlying infrastructure and the application owner's organization is responsible for the VM configuration, operating system, and any components installed on it.
  - **Modern applications** are hosted on PaaS environments such as an Azure application service. The underlying operating system is secured by the cloud provider. Application owners are responsible for the security of the application service configurations.
  - **Containers** are an application packaging mechanism in which applications are abstracted from the environment in which they run. The containerized applications can run on a container service by the cloud provider (modern applications) or on a server managed on premises or in IaaS.

Identify and classify applications

Identify applications that have a high potential impact and,or a high potential exposure to risk.

- **Business critical data**. Applications that process or store information must have assurance of confidentiality, integrity, and availability.
- **Regulated data**. Applications that handle monetary instruments and sensitive personal information regulated by standards such as the payment card industry (PCI), General Data Protection Regulation (GDPR), and Health Information Portability and Accountability Act (HIPAA).
- **Business critical availability**. Applications whose functionality is critical to the business mission, such as production lines generating revenue, devices or services critical to life and safety, and other critical functions.
- **Significant Access**. Applications that have access to systems with a high impact through technical means such as
  - Stored Credentials or keys/certificates that grant access to the data/service.
  - Permissions granted through access control lists or other methods.
- **High exposure to attacks**. Applications that are easily accessible to attackers such as web applications on the public internet. Legacy applications can also be higher exposure as attackers (and penetration testers) frequently target them because they know these legacy applications often have vulnerabilities that are difficult to fix.

Use Azure services for fundamental components

Developers should use services available from a cloud provider for well-established functions like databases, encryption, identity directory, and authentication, instead of building or adopting custom implementations, or third-party solutions that require integration with the cloud provider. These services provide better security, reliability, and efficiency because cloud providers operate and secure them with dedicated teams with deep expertise in those areas.

Using these services also frees your developer resources from reinventing the proverbial wheel so that they can focus development time on your unique requirements for your business. This practice should be followed to avoid risk during new application development and to reduce risk in existing applications either during the planned update cycle, or with a security-focused application update.

We recommend using cloud services from your cloud provider for identity, data protection, key management, and application configurations:

- **Identity:** User directories and other authentication functions are complex to develop and critically important to security assurances. Avoid custom authentication solutions. Instead choose native capabilities like Azure Active Directory (Azure AD), Azure AD B2B, Azure AD B2C, or third-party solutions to authenticate and grant permission to users, partners, customers, applications, services, and other entities. For more information, see Security with identity and access management (IAM) in Azure.

- **Data Protection:** Use established capabilities from cloud providers such as native encryption in cloud services to encrypt and protect data. If direct use of cryptography is required, use well-established cryptographic algorithms and not attempt to invent their own.
- **Key management:** Always authenticate with identity services rather than handling cryptographic key. For situations where you need to keys, use a managed key store such as Azure Key Vault. This will make sure keys are handled safely in application code. Tools such as, CredScan can discover potentially exposed keys in your application code.
- **Application Configurations:** Inconsistent configurations for applications can create security risks. Application configuration information can be stored with the application itself or preferably using a dedicated configuration management system like Azure App Configuration or Azure Key Vault. App Configuration provides a service to centrally manage application settings and feature flags, which helps mitigate this risk. Don't store keys and secrets in application configuration.

For more information about using cloud services instead of custom implementations, reference Applications and services.

Use native capabilities

Use native security capabilities built into cloud services instead of adding external security components, such as data encryption, network traffic filtering, threat detection, and other functions.

Azure controls are maintained and supported by Microsoft. You don't have to invest in additional security tooling.

- List of Azure Services
- Native security capabilities of each service

Next steps

- Applications and services
- Application classification
- Application threat analysis
- Regulatory compliance

Application threat analysis

11/19/2021 • 4 minutes to read • Edit Online

Do a comprehensive analysis to identify threats, attacks, vulnerabilities, and counter measures. Having this information can protect the application and threats it might pose to the system. Start with simple questions to gain insight into potential risks. Then, progress to advanced techniques using threat modeling.

1- Gather information about the basic security controls

A threat modeling tool will produce a report of all threats identified. This report is typically uploaded into a tracking tool, or converted to work items that can be validated and addressed by the developers. As new features are added to the solution, the threat model should be updated and integrated into the code management process. If a security issue is found, there should be a process to triage issue severity and determine when and how to remediate (such as in the next release cycle, or a faster release).

Start by gathering information about each component of the application. The answers to these questions will identify gaps in basic protection and clarify the attack vectors.

ASK THIS QUESTION ...	TO DETERMINE CONTROLS THAT ...
Are connections authenticated using Azure AD, TLS (with mutual authentication), or another modern security protocol approved by the security team? • Between users and the application • Between different application components and services	Prevent unauthorized access to the application component and data.
Are you limiting access to only those accounts that have the need to write or modify data in the application	Prevent unauthorized data tampering or alteration.
Is the application activity logged and fed into a Security Information and Event Management (SIEM) through Azure Monitor or a similar solution?	Detect and investigate attacks quickly.
Is critical data protected with encryption that has been approved by the security team?	Prevent unauthorized copying of data at rest.
Is inbound and outbound network traffic encrypted using TLS?	Prevent unauthorized copying of data in transit.
Is the application protected against Distributed Denial of Service (DDoS) attacks using services such as Azure DDoS protection?	Detect attacks designed to overload the application so it can't be used.
Does the application store any logon credentials or keys to access other applications, databases, or services?	Identify whether an attack can use your application to attack other systems.
Do the application controls allow you to fulfill regulatory requirements?	Protect user's private data and avoid compliance fines.

Suggested actions

Assign tasks to the individual people who are responsible for a particular risk identified during threat modeling.

Learn more

Threat modeling

2- Evaluate the application design progressively

Analyze application components and connections and their relationships. Threat modeling is a crucial engineering exercise that includes defining security requirements, identifying and mitigating threats, and validating those mitigations. This technique can be used at any stage of application development or production, but it's most effective during the design stages of a new functionality.

Popular methodologies include:

- STRIDE:
  - Spoofing
  - Tampering
  - Repudiation
  - Information Disclosure
  - Denial of Service
  - Elevation of Privilege

Microsoft Security Development Lifecycle uses STRIDE and provides a tool to assist with this process. This tool is available at no additional cost. For more information, see Microsoft Threat Modeling Tool.

- Open Web Application Security Project (OWASP) has documented a threat modeling approach for applications.

> Integrate threat modeling through automation using secure operations. Here are some resources:
>
> - Toolkit for Secure DevOps on Azure.
> - Guidance on DevOps pipeline security by OWASP.

3- Mitigate the identified threats

The threat modeling tool produces a report of all the threats identified. After a potential threat is identified, determine how it can be detected and the response to that attack. Define a process and timeline which minimizes exposure to any identified vulnerabilities in the workload, so that those vulnerabilities cannot be left unaddressed.

Use the *Defense-in-Depth* approach. This can help identify controls needed in the design to mitigate risk if a primary security control fails. Evaluate how likely it is for the primary control to fail. If it does, what is the extent of the potential organizational risk? Also, what is the effectiveness of the additional control (especially in cases that would cause the primary control to fail). Based on the evaluation apply Defense-in-Depth measures to address potential failures of security controls.

The principle of *least privilege* is one way of implementing Defense-in-Depth. It limits the damage that can be done by a single account. Grant least number of privileges to accounts that allows them to accomplish with the required permissions within a time period. This helps mitigate the damage of an attacker who gains access to the account to compromise security assurances.

There's often a disconnect between organizational leadership and technical teams regarding business requirements for critical workloads. This can create undesired outcomes and is especially sensitive when it pertains to information security. Routinely reviewing business critical workload requirements with executive

sponsors to define requirements provides an opportunity to align expectations and ensure operational resource allocation to the initiative.

How are threats addressed once found?

Here are some best practices:

- Make sure the results are communicated to the interested teams.
- Prioritize the vulnerabilities and fix the most important in a timely manner.
- Upload the threat modeling report to a tracking tool. Create work items that can be validated and addressed by the developers. Cyber security teams can also use the report to determine attack vectors during a penetration test.
- As new features are added to the application, update the threat model report and integrate it into the code management process. Triage security issues into the next release cycle or a faster release, depending on the severity.

For information about mitigation strategies, see RapidAttack.

How long does it typically take to deploy a security fix into production?

If a security vulnerability is discovered, update the software with the fix as soon as possible. Have processes, tools, and approvals in place to roll out the fix quickly.

Learn more

Threat modeling

Next steps

- Applications and services
- Application classification
- Regulatory compliance

Secure application configuration and dependencies

11/19/2021 • 3 minutes to read • Edit Online

Security of an application that is hosted in Azure is a shared responsibility between you as the application owner and Azure. For IaaS, you're responsible for configurations related to VM, operating system, and components installed on it. For PaaS, you're responsible for the security of the application service configurations and making sure that the dependencies used by the application are also secure.

Key points

- Don't store secrets in source code or configuration files. Instead, keep them in a secure store, such as Azure App Configuration or Azure Key Vault.
- Don't expose detailed error information when handling application exceptions.
- Don't expose platform-specific information.
- Store application configuration outside of the application code to update it separately and to have tighter access control.
- Restrict access to Azure resources that don't meet the security requirements.
- Validate the security of any open-source code added to your application.
- Update frameworks and libraries as part of the application lifecycle.

Configuration security

During the design phase, consider the way you store secrets and handle exceptions. Here are some points.

How is application configuration stored and how does the application access it?

Application configuration information can be stored with the application. However, that's not a recommended practice. Consider using a dedicated configuration management system such as Azure App Configuration or Azure Key Vault. That way, it can be updated independently of the application code.

Applications can include secrets like database connection strings, certificate keys, and so on. Don't store secrets in source code or configuration files. Instead, keep them in a secure store, such as Azure Key Vault. Identify secrets in code with static code scanning tools. Add the scanning process in your continuous integration (CI) pipeline.

For more information about secret management, reference Key and secret management.

Are errors and exceptions handled properly without exposing that information to users?

When handling application exceptions, make the application fail gracefully and log the error. Don't provide detailed information related to the failure, such as call stack, SQL queries, or out of range errors. This information can provide attackers with valuable information about the internals of the application.

Can configuration settings be changed or modified without rebuilding or redeploying the application?

Application code and configuration shouldn't share the same lifecycle to enable operational activities. These activities include those that change and update specific configurations without developer involvement or redeployment.

Is platform-specific information removed from server-client communication?

Don't reveal information about the application platform. Such information (for example, `X-Powered-By`, `X-ASPNET-VERSION`) can get exposed through HTTP banners, HTTP headers, error messages, and website footers. Malicious actors can use this information when mapping attack vectors of the application.

Suggested actions

Consider using Azure Front Door or API Management to remove platform-specific HTTP headers. Instead, use Azure CDN to separate the hosting platform from end users. Azure API Management offers transformation policies that allow you to modify HTTP headers and remove sensitive information.

Learn more

- Azure Front Door Rules Engine Actions
- API Management documentation

Are Azure policies used to control the configuration of the solution resources?

Use Azure Policy to deploy settings where applicable. Block resources that don't meet the proper security requirements defined during service enablement.

Dependencies, frameworks, and libraries

What are the frameworks and libraries used by the application?

Application frameworks are frequently updated and released by the vendor or communities. It's vital to track the frameworks and libraries used by the application including any resulting vulnerabilities they introduce. These frameworks and libraries include custom, OSS, third party, and others. Understand and manage the technologies the application uses, such as:

- .NET Core
- Spring
- Node.js

Automated solutions can help with this assessment.

Consider the following best practices:

- Validate the security of any open-source code added to your application. Free tools to help with this assessment include:
  - OWASP Dependency-Check
  - NPM audit
  - WhiteSource Bolt

  These tools find outdated components and update them to the latest versions.
- Maintain a list of frameworks and libraries as part of the application inventory. Also, keep track of versions in use. If vulnerabilities are published, this awareness helps to identify affected workloads.
- Update frameworks and libraries as part of the application lifecycle. Prioritize critical security patches.

Are the expiry dates of SSL/TLS certificates monitored and are processes in place to renew them?

Tracking expiry dates of SSL/TLS certificates and renewing them in due time is highly critical. Ideally, the process should be automated, although this often depends on the CA used for the certificate. If not automated, sufficient alerting should be applied to ensure expiry dates don't go unnoticed.

Learn more

- WhiteSource Bolt
- npm-audit
- OWASP Dependency-Check

Referenced Azure services

- Azure Key Vault
- Azure CDN
- Azure Policy

Next steps

- Applications and services
- Application classification
- Application threat analysis
- Regulatory compliance

Community resources

- OWASP Dependency-Check
- NPM audit

Secure deployment in Azure

11/19/2021 • 2 minutes to read • Edit Online

Have teams, processes, and tools that can quickly deploy security fixes? A *DevOps* or multidisciplinary approach is recommended. Multiple teams work together with efficient practices and tools. Essential DevOps practices include change management of the workload through continuous integration, continuous delivery (CI/CD).

Continuous integration (CI) is an automated process where code changes trigger the building and testing of the application. Continuous Delivery (CD) is an automated process to build, test, configure, and deploy the application from a build to production environment.

Those processes allow you to rapidly address the security concerns without waiting for a longer planning and testing cycle.

Building a DevOps process which includes a security discipline helps incorporate security concepts and enhancements earlier in the application development process. An organization's ability to rapidly address security and operational concerns increases through the combination of the Secure Development Lifecycle (SDL) and Operations Lifecycle related to application creation, maintenance, and updates.

Many traditional IT operating models aren't compatible with the cloud, and organizations must undergo operational and organizational transformation to deliver against enterprise migration targets. We recommend using a DevOps approach for both application and central teams.

Checklist

Have you adopted a secure DevOps approach to ensure security and feature enhancements can be quickly deployed?

- Establish a cross-functional DevOps platform team to build, manage, and maintain your workload.
- Involve the security team in the planning and design of the DevOps process to integrate preventive and detective controls for security risks.
- Clearly define CI/CD roles and permissions and minimize the number of people who have access to secure information or resources.
- Configure quality gate approvals in DevOps release process.

- Integrate scanning tools within CI/CD pipeline.
- No infrastructure changes, provisioning or configuring, should be done manually outside of IaC.

In this section

Follow these questions to assess the workload at a deeper level.

ASSESSMENT	DESCRIPTION
Do you clearly define CI/CD roles and permissions for this workload?	Define CI/CD permissions such that only users responsible for production releases can start the process and that only developers can access the source code.
Are any resources provisioned or operationally configured with user tools such as the Azure portal or via Azure CLI?	Always use Infrastructure as code (IaC) to make even the smallest of changes. This approach makes it easy to track code because the provisioned infrastructure is reproducible and reversible.
Can you roll back or forward code quickly through automated pipelines?	Automated deployment pipelines should allow for quick roll-forward and roll-back deployments to address critical bugs and code updates outside of the normal deployment lifecycle.

Azure security benchmark

The Azure Security Benchmark includes a collection of high-impact security recommendations. Use them to secure the services and processes you use to run the workload in Azure:

> The questions in this section are aligned to the Azure Security Benchmark controls.

Reference architecture

Here are some reference architectures related to building CI/CD pipelines:

- CI/CD for microservices architectures
- CI/CD for microservices on Kubernetes

Next step

We recommend monitoring activities that maintain the security posture. These activities can highlight, if the current security practices are effective or are there new requirements.

Security monitoring

Related link

> Go back to the main article: Security

Learn more

- Secure DevOps Kit for Azure
- Agile Principles in Practice
- Platform automation and DevOps

Governance considerations for secure deployment in Azure

11/19/2021 • 4 minutes to read • Edit Online

The automated continuous integration, continuous delivery (CI/CD) processes must have built-in governance that authorize and authenticate the identities to do the tasks within a defined scope.

Key points

- Clearly define CI/CD roles and permissions.
- Implement just-in-time privileged access management.
- Limit long-standing write access to production environments.
- Limit the scope of execution in the pipelines.
- Configure quality gate approvals in DevOps release process.

Minimize access

Minimize the number of people who have access to secure information or resources. This strategy will reduce the chance of a malicious actor gaining access or an authorized user inadvertently impacting a sensitive resource. Here are some considerations:

- Use the principle of least privilege when assigning roles and permissions. Only users responsible for production releases should start the process and only developers should access the source code.

  A pipeline should use one or more service principals. Ideally, they should be managed identity, and delivered by the platform and never directly defined within a pipeline. The identity should only have the Azure RBAC permissions necessary to do the task. All service principals should be bound to that pipeline and not shared across pipelines.

  How do you define CI/CD roles and permissions?

  Azure DevOps offers built-in roles that can be assigned to individual users of groups. If built-in roles are insufficient to define least privilege for a pipeline, consider creating custom Azure RBAC roles. Make sure those roles align with the action and the organization's teams and responsibilities.

  To support security of your pipeline operations, you can add users to a built-in security group, set individual permissions for a user or group, or add users to pre-defined roles. You manage security for the following objects from Azure Pipelines in the web portal, either from the user or admin context.

  For more information, see Get started with permissions, access, and security groups.

- For permissions, you grant or restrict permissions by setting the permission state to **Allow** or **Deny**, either for a security group or an individual user. For a role, you add a user or group to the role.

- Use separate pipeline identities between pre-production and production environments. If available, take advantage of pipeline features such as Environments to encapsulate last-mile authentication external to the executing pipeline.

- If the pipeline runs infrequently and has high privileges, consider removing standing permissions for that identity. Use just-in-time (JIT) role assignments, time-based, and approval-based role activation. This strategy will mitigate the risks of excessive, unnecessary, or misused access permissions on crucial

resources. Azure AD Privileged Identity Management supports all those modes of activation.

- Review the organization's CI/CD pipeline and refine role assignment to create a clear delineation between development and production responsibilities.

Learn more

For more information about pipeline permission and security roles, reference Set different levels of pipeline permissions.

Execution scope

Where practical, limit the scope of execution in the pipelines.

Consider creating a multi-stage pipeline. Divide the work into discrete units and that can be isolated in a separate pipeline. Limit the identities only to the scope of the unit so that it has minimal privileges enough to do the action. For example, you can have two units, one to deploy and another that builds source code. Only allow the deploy unit to have access to the identity, not the build unit. If the build unit is compromised, it could start tampering with the infrastructure.

Gated approval process

Do you have release gate approvals configured in the DevOps release process?

Pull Requests and code reviews serve as the first line of approvals during development cycle. Before releasing an update to production, require a process that mandates security review and approval.

Make sure that you involve the security team in the planning, design, and DevOps process. This collaboration will help them implement security controls, auditing, and response processes.

Are branch policies used in source control management of this workload? How are they configured?

Establish branch policies that provide an extra level of control over the code that is committed to the repository. Lack of secure branch policy might allow poor, rogue or broken code to be checked-in and deployed. It's a common practice to deny pushes to the main branch if the change isn't approved. For example, you can require pull-request (PR) with code review before merging the changes by at least one reviewer, other than the change author.

Having multiple branches is recommended where each branch has a purpose and access level. For example, feature branches are created by developers and are open to push. Integration branch requires PR and code-review. Production branch requires another approval from the team lead before merging.

Suggested actions

- Configure quality gate approvals in DevOps release process.
- Follow the guidance in the linked articles to deploy and adopt branch strategy.

Learn more

- About branches and branch policies
- Adopt a Git branching strategy
- Release deployment control using gates

Next

Secure infrastructure deployments

Related links

Go back to the main article: Secure deployment and testing in Azure

Infrastructure provisioning considerations in Azure

11/19/2021 • 2 minutes to read • Edit Online

Azure resources can be provisioned in through code or user tools such as the Azure portal or via Azure CLI. It's not recommended that resources are provisioned or configured manually. Those methods are error prone and can lead to security gaps. Even the smallest of changes should be through code. The recommended approach is Infrastructure as code (IaC). It's easy to track because the provisioned infrastructure can be fully reproduced and reversed.

Key points

- No infrastructure changes should be done manually outside of IaC.
- Store keys and secrets outside of deployment pipeline in Azure Key Vault or in secure store for the pipeline.
- Incorporate security fixes and patching to the operating system and all parts of the codebase, including dependencies (preinstalled tools, frameworks, and libraries).

Infrastructure as code (IaC)

Make all operational changes and modifications through IaC. This is a key DevOps practice, and it's often used with continuous delivery. IaC manages the infrastructure - such as networks, virtual machines, and others - with a descriptive model, using a versioning system that is similar to what is used for source code. IaC model generates the same environment every time it's applied. Common examples of IaC are Azure Resource Manager or Terraform.

IaC reduces configuration effort and automates full environment deployment (production and pre-production). Also, IaC allows you to develop and release changes faster. All those factors enhance the security of the workload.

For detailed information about IaC, see What is Infrastructure as code (IaC).

Pipeline secret management

How are credentials, certificates, and other secrets used in the operations for the workload managed during deployment?

Store keys and secrets outside of deployment pipeline in a managed key store, such as Azure Key Vault. Or, in a secure store for the pipeline. When deploying application infrastructure with Azure Resource Manager or Terraform, the process might generate credentials and keys. Store them in a managed key store and make sure the deployed resources reference the store. Do not hard-code credentials.

Build environments

Does the organization apply security controls (IP firewall restrictions, update management) to self-hosted build agents for this workload?

Custom build agents add management complexity and can become an attack vector. Build machine credentials must be stored securely and the file system needs to be cleaned of any temporary build artifacts regularly. Network isolation can be achieved by only allowing outgoing traffic from the build agent, because it's using the pull model of communication with Azure DevOps.

As part of the operational lifecycle, incorporate security fixes and patching to the operating system and all parts

of the codebase, including dependencies (preinstalled tools, frameworks, and libraries).

Apply security controls to self-hosted build agents in the same manner as with other Azure IaaS VMs. These should be minimalistic environments as a way to reduce the attack surface.

Learn more

- Azure Pipelines agents
- I'm running a firewall and my code is in Azure Repos. What URLs does the agent need to communicate with?

Next step

Secure code deployments

Related links

Go back to the main article: Secure deployment and testing in Azure

Code deployments

11/19/2021 • 4 minutes to read • Edit Online

The automated build and release pipelines should update a workload to a new version seamlessly without breaking dependencies. Augment the automation with processes that allow high priority fixes to get deployed quickly.

Organizations should leverage existing guidance and automation when securing applications in the cloud, rather than starting from zero. Using resources and lessons learned by external organizations that are early adopters of these models can accelerate the improvement of an organizations security posture with less expenditure of effort and resources.

Key points

- Involve the security team in the planning and design of the DevOps process to integrate preventive and detective controls for security risks.
- Design automated deployment pipelines that allow for quick roll-forward and rollback deployments to address critical bugs and code updates outside of the normal deployment lifecycle.
- Integrate code scanning tools within CI/CD pipeline.

Rollback and roll-forward

If something goes wrong, the pipeline should roll back to a previous working version. N-1 and N+1 refer to rollback and roll-forward versions. Automated deployment pipelines should allow for quick roll-forward and rollback deployments to address critical bugs and code updates outside of the normal deployment lifecycle.

Can N-1 or N+1 versions be deployed via automated pipelines where N is current deployment version in production?

Because security updates are a high priority, design a pipeline that supports regular updates and critical security fixes.

A release is typically associated with approval processes with multiple sign-offs, quality gates, and so on. If the workload deployment is small with minimal approvals, you can usually use the same process and pipeline to release a security fix.

An approval process that is complex and takes a significant amount of time can delay a fix. Consider building an emergency process to accelerate high priority fixes. The process might be business and, or communication process between teams. Another way is to build a pipeline that might not include all the gated approvals, but should be able to push out the fix quickly. The pipeline should allow for quick roll-forward and rollback deployments that address security fixes, critical bugs, and code updates outside of the regular deployment life cycle.

> **IMPORTANT**
>
> Deploying a security fix is a priority, but it shouldn't be at the cost of introducing a regression or bug. When designing an emergency pipeline, carefully consider which automated tests can be bypassed. Evaluate the value of each test against the execution time. For example, unit tests usually complete quickly. Integration or end-to-end tests can run for a long time.

Involve the security team in the planning and design of the DevOps process. Your automated pipeline design

should have the flexibility to support both regular and emergency deployments. This is important to support the rapid and responsible application of both security fixes and other urgent, important fixes.

Suggested action

Implement an automated deployment process with support for rollback scenarios via Azure App Services deployment slots.

Learn more

Set up staging environments in Azure App Service

Credential scanning

Credentials, keys, and certificates grant access to the data or service used by the workload. Storing credentials in code poses a significant security risk. Ensure that static code scanning tools are an integrated part of the continuous integration (CI) process.

Are code scanning tools an integrated part of the continuous integration (CI) process for this workload?

To prevent credentials from being stored in the source code or configuration files, integrate code scanning tools within the CI/CD pipeline:

- During design time, use code analyzers to prevent credentials from getting pushed to the source code repository. For example, .NET Compiler Platform (Roslyn) Analyzers inspect your C# or Visual Basic code.
- During the build process, use pipeline add-ons to catch credentials in the source code. Some options include GitHub Advanced Security and OWASP source code analysis tools.
- Scan all dependencies, such as third-party libraries and framework components, as part of the CI process. Investigate vulnerable components that are flagged by the tool. Combine this task with other code scanning tasks that inspect code churn, test results, and coverage.
- Use a combination of dynamic application security testing (DAST) and static application security testing (SAST). DAST tests the application while its in use. SAST scans the source code and detects vulnerabilities based on its design or implementation. Some technology options are provided by OWASP. For more information, see SAST Tools and Vulnerability Scanning Tools.
- Use scanning tools that are specialized in technologies used by the workload. For example, if the workload is containerized, run container-aware scanning tools to detect risks in the container registry, before use, and during use.

Suggested actions

Incorporate Secure DevOps on Azure toolkit and the guidance published by the Organization for Web App Security Project (OWASP), or an equivalent guiding organization.

Learn more

- Follow DevOps security guidance
- Getting started with Credential Scanner (CredScan)

Community links

- OWASP source code analysis tools
- GitHub Advanced Security
- Vulnerability Scanning Tools

Go back to the main article: Secure deployment and testing in Azure

Security monitoring and remediation in Azure

11/19/2021 • 2 minutes to read • Edit Online

Regularly monitor resources to maintain the security posture and detect vulnerabilities. Detection can take the form of reacting to an alert of suspicious activity or proactively hunting for anomalous events in the enterprise activity logs. vigilantly responding to anomalies and alerts to prevent security assurance decay, and designing for defense in depth and least privilege strategies.

Checklist

How are you monitoring security-related events in this workload?

- Use native tools in Azure to monitor the workload resources and the infrastructure in which it runs.
- Consider investing in a Security Operations Center (SOC), or SecOps team and incident response plan.
- Monitor traffic, access requests, and application communication between segments.
- Discover and remediate common risks to improve secure score in Microsoft Defender for Cloud.
- Use an industry standard benchmark to evaluate the security posture by learning from external organizations.
- Send logs and alerts to a central security log management for analysis.
- Perform regular internal and external compliance audits, including regulatory compliance attestations.
- Regularly test your security design and implementation using test cases based on real-world attacks.

Azure security benchmark

The Azure Security Benchmark includes a collection of high-impact security recommendations. Use them to secure the services and processes you use to run the workload in Azure:

> The questions in this section are aligned to these controls:
>
> - Azure Security Benchmarks Logging and threat detection.
> - Azure Security Benchmarks Incident response.
> - Posture and Vulnerability Management

Reference architecture

- Hybrid Security Monitoring using Microsoft Defender for Cloud and Microsoft Sentinel

  This reference architecture illustrates how to use Microsoft Defender for Cloud and Microsoft Sentinel to monitor the security configuration and telemetry of on-premises and Azure operating system workloads.

- Azure security solutions for AWS

  This article provides AWS identity architects, administrators, and security analysts with immediate insights and detailed guidance for deploying several Microsoft security solutions.

Next step

We recommend applying as many best practices as early as possible, and then working to retrofit any gaps over time as you mature your security program.

Related link

Go back to the main article: Security

Azure security monitoring tools

11/19/2021 • 5 minutes to read • Edit Online

The *leverage native control* security principle tells us to use native controls built into cloud services over external controls through third-party solutions. Native reduce the effort required to integrate external security tooling and update those integrations over time.

Azure provides several monitoring tools that observe the operations and detect anomalous behavior. These tools can detect threats at different levels and report issues. Addressing the issues early in the operational lifecycle will strengthen your overall security posture.

Tools

SERVICE	USE CASE
Microsoft Defender for Cloud	Strengthens the security posture of your data centers, and provides advanced threat protection across your workloads in the cloud (whether they're in Azure or not) and on-premises. Get a unified view into the infrastructure and resources provisioned for the workload.
Microsoft Sentinel	Use the native security information event management (SIEM) and security orchestration automated response (SOAR) solution on Azure. Receive intelligent security analytics and threat intelligence across the enterprise.
Azure DDoS Protection	Defend against distributed denial of service (DDoS) attacks.
Azure Rights Management (RMS)	Protect files and emails across multiple devices.
Microsoft Information Protection (MIP)	Secure email, documents, and sensitive data that you share outside your company.
Azure Governance Visualizer	Gain granular insight into policies, Azure role-based access control (Azure RBAC), Azure Blueprints, subscriptions, and more.

Microsoft Defender for Cloud

Enable Microsoft Defender for Cloud at the subscription level to monitor all resource provisioned in that scope. At no additional cost, it provides continuous observability into resources, reports issues, and recommends fixes. By regularly reviewing and fixing issues, you can improve the security posture, detect threats early, prevent breaches.

Beyond just observability, Defender for Cloud offers an advanced mode through its integration with **Microsoft Defender for Cloud**. When these plans are enabled, built-in policies, custom policies, and initiatives protect resources and block malicious actors. You can also monitor compliance with regulatory standards - such as NIST, Azure CIS, Azure Security Benchmark. For pricing details, see Defender for Cloud pricing.

Microsoft Sentinel

Your organization might run workloads on multiple cloud platforms, and, or across cloud and on-premises, or managed by various teams within the organization. Having a centralized view of all data is recommended. To get that view you need security information event management (SIEM) and security orchestration automated response (SOAR) solutions. These solutions connect to all security sources, monitor them, and analyze the correlated data.

Microsoft Sentinel and is a native control that combines SIEM and SOAR capabilities. It analyzes events and logs from various connected sources. Based on the data sources and their alerts, Sentinel creates incidents, performs threat analysis for early detection. Through intelligent analytics and queries, you can be proactive with hunting activities. In case of incidents, you can automate workflows. Also, with workbook templates you can quickly gain insights through visualization.

Azure DDoS Protection

A Distributed Denial of Service (DDoS) attack attempts to exhaust an application's resources, making the application unavailable to legitimate users. DDoS attacks can be targeted at any endpoint that is publicly reachable through the internet.

Every property in Azure is protected by Azure's infrastructure DDoS (Basic) Protection at no additional cost. The scale and capacity of the globally deployed Azure network provides defense against common network-layer attacks through always-on traffic monitoring and real-time mitigation. DDoS Protection Basic requires no user configuration or application changes. DDoS Protection Basic helps protect all Azure services, including PaaS services like Azure DNS.

Azure DDoS Protection Standard provides enhanced DDoS mitigation features to defend against DDoS attacks. It is automatically tuned to help protect your specific Azure resources in a virtual network. Protection is simple to enable on any new or existing virtual network, and it requires no application or resource changes. It has several advantages over the basic service, including logging, alerting, telemetry, SLA guarantee and cost protection.

Azure DDoS Protection Standard is designed for services that are deployed in a virtual network. For other services, the default DDoS Protection Basic service applies. To learn more about supported architectures, see DDoS Protection reference architectures.

Azure Rights Management (RMS)

Your business may encounter challenges with protecting documents and emails. For example, file protection, collaboration, and sharing may be issues. You also might be experiencing problems regarding platform support or infrastructure.

Azure Rights Management (RMS) is a cloud-based protection service that uses encryption, identity, and authorization policies to help secure files and emails across multiple devices, including phones, tablets, and PCs.

To learn more about how RMS can address these issues, see Business problems solved by Azure Rights Management.

Microsoft Information Protection (MIP)

The *data classification* process categorizes data by sensitivity and business impact in order to identify risks. When data is classified, you can manage it in ways that protect sensitive or important data from theft or loss.

With proper *file protection*, you can analyze data flows to gain insight into your business, detect risky behaviors and take corrective measures, track access to documents, and more. The protection technology in AIP uses encryption, identity, and authorization policies. Protection stays with the documents and emails, independently of the location, regardless of whether they are inside or outside your organization, networks, file servers, and applications

Azure Information Protection (AIP) is part of Microsoft Information Protection (MIP) solution, and extends the labeling and classification functionality provided by Microsoft 365. For more information, see this article about classification.

Azure Governance Visualizer

Azure Governance Visualizer is a PowerShell script that iterates through an Azure tenant's management group hierarchy down to the subscription level. You can run the script either for your Tenant Root Group or any other Management Group. It captures data from the most relevant Azure governance capabilities such as Azure Policy, Azure role-based access control (Azure RBAC), and Azure Blueprints. From the collected data, the visualizer shows your hierarchy map, creates a tenant summary, and builds granular scope insights about your management groups and subscriptions.

The visualizer provides a holistic overview of your technical Azure Governance implementation by connecting the dots.

Next

Monitor workload resources in Microsoft Defender for Cloud

Related links

For information on the Microsoft Defender for Cloud tools, see Strengthen security posture.

For frequently asked questions on Microsoft Defender for Cloud, see FAQ - General Questions.

For information on the Microsoft Sentinel tools that will help to meet these requirements, see What is Microsoft Sentinel?

For types of DDoS attacks that DDoS Protection Standard mitigates as well as more features, see Azure DDoS Protection Standard overview.

Monitor Azure resources in Microsoft Defender for Cloud

11/19/2021 • 10 minutes to read • Edit Online

Most cloud architecture have compute, networking, data, and identity components and each require different monitoring mechanisms. Even Azure services have individual monitoring needs. For instance, to monitor Azure Functions you want to enable Azure Application Insights.

Microsoft Defender for Cloud has many plans that monitor the security posture of machines, networks, storage and data services, and applications to discover potential security issues. Common issues include internet connected VMs, or missing security updates, missing endpoint protection or encryption, deviations from baseline security configurations, missing Web Application Firewall (WAF), and more.

Key points

[!div class="checklist"]

- Enable Microsoft Defender for Cloud as a defense-in-depth measure. Use resource-specific Defender for Cloud features such as Microsoft Defender for servers, Microsoft Defender for Endpoint, Microsoft Defender for Storage.
- Observe container hygiene through container aware tools and regular scanning.
- Review all network flow logs through network watcher. See diagnostic logs in Microsoft Defender for Cloud.
- Integrate all logs in a central SIEM solution to analyze and detect suspicious behavior.
- Monitor identity-related risk events in Azure AD reporting amd Azure Active Directory Identity Protection.

General best practices

Identifying common security activities will significantly reduce the overall risk.

- Monitor suspicious activities from administrative accounts.
- Monitor the location from where Azure resources are being managed.
- Monitor attempts to access deactivated credentials.
- Use automated tools to monitor network resource configurations and detect changes.

For more information, see Azure security baseline for Azure Monitor.

IaaS and PaaS security

In an IaaS model, you can host the workload on Azure infrastructure. Azure provides security assurances that maintain isolation and timely security updates to the infrastructure. For greater control, you host the entire IaaS solution on-premises or in a hosted data center and are responsible for security. You must implement security on the host, virtual machine, network, and storage. For instance if you have your own VNet, consider enabling Azure Private Link over Azure Monitor so you can access this over a private endpoint.

In PaaS, you have shared responsibility with Azure in protecting the data.

Virtual machines

If you're running your own Windows and Linux virtual machines, use Microsoft Defender for Cloud. Take advantage of the free services to check for missing OS patches, security misconfiguration, and basic network security. Enabling Microsoft Defender for Cloud is highly recommended because you get features that provide adaptive application controls, file integrity monitoring (FIM), and others.

For example, a common risk is the virtual machines don't have vulnerability scanning solutions that check for threats. Microsoft Defender for Cloud reports those machines. You can remediate in Microsoft Defender for Cloud by deploying a scanning solution. You can use the built-in vulnerability scanner for virtual machines. You don't need a license. Instead, you can bring your license for supported partner solutions.

> **NOTE**
> Vulnerability assessments are also available for container images, and SQL servers.

Attackers constantly scan public cloud IP ranges for open management ports, which can lead to attacks such as common passwords and known unpatched vulnerabilities. JIT (Just In Time) access allows you to lock down the inbound traffic to the virtual machines while providing easy access to connect to machines when needed. Defender for Cloud identifies which machines should have JIT applied.

With Microsoft Defender for Cloud, you also get Microsoft Defender for Endpoint. This provides investigative tools Endpoint Detection and Response (EDR) that helps in threat detection and analysis.

Microsoft Defender for servers also watches the network to and from virtual machines. If you are using network security groups to control access to the virtual machines and the rules are overpermissive, Defender for Cloud will flag them. Adaptive network hardening provides recommendations to further harden the NSG rules.

For a full list of features, see Feature coverage for machines.

Remove direct internet connectivity

Make sure policies and processes require restricting and monitoring direct internet connectivity by virtual machines.

For Azure, you can enforce policies by:

- **Enterprise-wide prevention:** Prevent inadvertent exposure by following the permissions and roles described in the reference model.
  - Ensures that network traffic is routed through approved egress points by default.
  - Exceptions (such as adding a public IP address to a resource) must go through a centralized group that evaluates exception requests and makes sure appropriate controls are applied.
- **Identify and remediate** exposed virtual machines by using the Microsoft Defender for Cloud network visualization to quickly identify internet exposed resources.
- **Restrict management ports** (RDP, SSH) using Just in Time access in Microsoft Defender for Cloud.

One way of managing VMs in the virtual network is by using Azure Bastion. This service allows you to log into VMs in the virtual network through SSH or remote desktop protocol (RDP) without exposing the VMs directly to the internet. To see a reference architecture that uses Bastion, see Network DMZ between Azure and an on-premises datacenter.

Containers

Containerized workloads have an extra layer of abstraction and orchestration. That complexity requires specific security measures that protect against common container attacks such as supply chain attacks.

- Use container registries that are validated for security. Images in public registries might contain malware or unwanted applications that activate when the container is running. Build a process for developers to request and rapidly get security validation of new containers and images. The process should validate against your security standards. This includes applying security updates, scanning for unwanted code such as backdoors and illicit crypto coin miners, scanning for security vulnerabilities, and application of secure development practices.

  A popular process pattern is the quarantine pattern. This pattern allows you to get your images on a dedicated container registry and subject them to security or compliance scrutiny applicable for your organization. After it's validated, they can then be released from quarantine and promoted to being available.

  Microsoft Defender for Cloud identifies unmanaged containers hosted on IaaS Linux VMs, or other Linux machines running Docker containers.

- Make sure you use images from authorized registries. You can enforce this restriction through Azure Policy. For example, for an Azure Kubernetes Service (AKS) cluster, have policies that restrict the cluster to only pull images from Azure Container Registry (ACR) that is deployed as part of the architecture.

  > **TIP**
  >
  > Here are the resources for the preceding example:
  >
  > GitHub: Azure Kubernetes Service (AKS) Secure Baseline Reference Implementation.
  >
  > The design considerations are described in Baseline architecture for an AKS cluster.

- Regularly scan containers for known risks in the container registry, before use, and during use.

- Use security monitoring tools that are container aware to monitor for anomalous behavior and enable investigation of incidents.

  Microsoft Defender for container registries are designed to protect AKS clusters, container hosts (virtual machines running Docker), and ACR registries. When enabled, the images that are pulled or pushed to registries are subject to vulnerability scans.

For more information, see these articles:

- Container security in Defender for Cloud

Network

How do you monitor and diagnose conditions of the network?

As an initial step, enable and review all logs (including raw traffic) from your network devices.

- Security group logs – flow logs and diagnostic logs
- Azure Network Watcher

Take advantage of the packet capture feature to set alerts and gain access to real-time performance information at the packet level.

Packet capture tracks traffic in and out of virtual machines. It gives you the capability to run proactive captures based on defined network anomalies including information about network intrusions.

For an example, see Scenario: Get alerts when VM is sending you more TCP segments than usual.

Then, focus on observability of specific services by reviewing the diagnostic logs. For example, for Azure Application Gateway with integrated WAF, see Web application firewall logs. Microsoft Defender for Cloud analyzes diagnostic logs on virtual networks, gateways, network security groups and determines if the controls are secure enough. For example:

- Is your virtual machine exposed to public internet. If so, do you have tight rules on network security groups to protect the machine?
- Are the network security groups (NSG) and rules that control access to the virtual machines overly permissive?
- Are the storage accounts receiving traffic over secure connections?

Follow the recommendations provided by Defender for Cloud. For more information, see Networking recommendations. Use Azure Firewall logs and metrics for observability into operational and audit logs.

Integrate all logs into a security information and event management (SIEM) service, such as Microsoft Sentinel. The SIEM solutions support ingestion of large amounts of information and can analyze large datasets quickly. Based on those insights, you can:

- Set alerts or block traffic crossing segmentation boundaries.
- Identify anomalies.
- Tune the intake to significantly reduce the false positive alerts.

Identity

Monitor identity-related risk events using adaptive machine learning algorithms, heuristics quickly before the attacker can gain deeper access into the system.

Review identity risks

Most security incidents take place after an attacker initially gains access using a stolen identity. Even if the identity has low privileges, the attacker can use it to traverse laterally and gain access to more privileged identities. This way the attacker can control access to the target data or systems.

Does the organization actively monitor identity-related risk events related to potentially compromised identities?

Monitor identity-related risk events on potentially compromised identities and remediate those risks. Review the reported risk events in these ways:

- Azure AD reporting. For information, see users at risk security report and the risky sign-ins security report.
- Use the reporting capabilities of Azure Active Directory Identity Protection.
- Use the Identity Protection risk events API to get programmatic access to security detections by using Microsoft Graph. See riskDetection and riskyUser APIs.

Azure AD uses adaptive machine learning algorithms, heuristics, and known compromised credentials (username/password pairs) to detect suspicious actions that are related to your user accounts. These username/password pairs come from monitoring public and dark web and by working with security researchers, law enforcement, security teams at Microsoft, and others.

Remediate risks by manually addressing each reported account or by setting up a user risk policy to require a password change for high risk events.

Regularly review critical access

Regularly review roles that are assigned privileges with a business-critical impact.

Set up a recurring review pattern to ensure that accounts are removed from permissions as roles change. You can conduct the review manually or through an automated process by using tools such as Azure AD access

reviews.

Discover & replace insecure protocols

Discover and disable the use of legacy insecure protocols SMBv1, LM/NTLMv1, wDigest, Unsigned LDAP Binds, and Weak ciphers in Kerberos.

Applications should use the SHA-2 family of hash algorithms (SHA-256, SHA-384, SHA-512). Use of weaker algorithms, like SHA-1 and MD5, should be avoided.

Authentication protocols are a critical foundation of nearly all security assurances. These older versions can be exploited by attackers with access to your network and are often used extensively on legacy systems on Infrastructure as a Service (IaaS).

Here are ways to reduce your risk:

- Discover protocol usage by reviewing logs with Microsoft Sentinel's Insecure Protocol Dashboard or third-party tools.
- Restrict or Disable use of these protocols by following guidance for SMB, NTLM, WDigest.
- Use only secure hash algorithms (SHA-2 family).

We recommend implementing changes using pilot or other testing methods to mitigate risk of operational interruption.

Learn more

For more information about hash algorithms, see Hash and Signature Algorithms.

Connected tenants

Does your security team have visibility into all existing subscriptions and cloud environments? How do they discover new ones?

Make sure the security team is aware of all enrollments and associated subscriptions connected to your existing environment through ExpressRoute or Site-Site VPN. Monitor them as part of the overall enterprise.

Assess if organizational policies and applicable regulatory requirements are followed for the connected tenants. This applies to all Azure environments that connect to your production environment network.

The organizations' cloud infrastructure should be well documented, with security team access to all resources required for monitoring and insight. Conduct frequent scans of the cloud-connected assets to ensure no additional subscriptions or tenants have been added outside of organizational controls. Regularly review Microsoft guidance to ensure security team access best practices are consulted and followed.

For information about permissions for this access, see Assign privileges for managing the environment section.

Suggested actions

Ensure all Azure environments that connect to your production environment and network apply your organization's policy, and IT governance controls for security.

You can discover existing connected tenants using a tool provided by Microsoft. Guidance on permissions you may assign to security is in the Assign privileges for managing the environment section.

CI/CD pipelines

DevOps practices are for change management of the workload through continuous integration, continuous delivery (CI/CD). Make sure you add security validation in the pipelines. Follow the guidance described in Learn how to add continuous security validation to your CI/CD pipeline.

Next steps

View logs and alerts

Security logs and alerts using Azure services

11/19/2021 • 5 minutes to read • Edit Online

Logs provide insight into the operations of a workload, the infrastructure, network communications, and so on. When suspicious activity is detected, use alerts as a way of detecting potential threats. As part of your defense-in-depth strategy and continuous monitoring, respond to the alerts to prevent security assurance from decaying over time.

Key points

- Configure central security log management.
- Enable audit logging for Azure resources.
- Collect security logs from operating systems.
- Configure security log storage retention.
- Enable alerts for anomalous activities.

Use native services

- **Azure Monitor** provides observability across your entire environment. You automatically get platform metrics, activity logs, and diagnostics logs from most of your Azure resources with no configuration. The activity logs provide detailed diagnostic and auditing information.
- **Microsoft Defender for Cloud** generates notifications as security alerts by collecting, analyzings, and integrating log data from your Azure resources and the network. Alerts are available when you enable the Microsoft Defender plans. This will add to the overall cost.
- **Microsoft Sentinel** is a security information event management (SIEM) and security orchestration automated response (SOAR) solution. It's a single solution for alert detection, threat visibility, proactive hunting, and threat response.

Ideally use a combination of the preceding services to get a full view. For example, use Azure Monitor to collect information about the operating system running on Azure compute. If you're running your own compute, use Microsoft Defender for Cloud.

Audit logging

An important aspect of monitoring is tracking operations. For example, you want to know who created, updated, deleted a resource. Or, get resource-specific information such as when an image was pulled from Azure Container Registry. That information is crucial for a Security Operations (SecOps) team in detecting the presence of adversaries, reacting to an alert of suspicious activity, or proactively hunting for anomalous events. They are also useful for security auditing and compliance and offline analysis.

On Azure, that information is emitted as platform logs by the resources and the platform on which they run. They are tracked by Azure Resource Manager as and when subscription-level events occur. Each resource emits logs specific to the service.

Consider storing your data for audit purposes or statistical analysis. You can retain data in your log analytics workspace and specify the data type. This example sets the retention for `SecurityEvents` to 730 days:

```
PUT /subscriptions/00000000-0000-0000-0000-
00000000000/resourceGroups/MyResourceGroupName/providers/Microsoft.OperationalInsights/workspaces/MyWorkspac
eName/Tables/SecurityEvent?api-version=2017-04-26-preview {"properties":  {"retentionInDays": 730 } }
```

Retaining data in this manner can reduce your costs for data retention over time. For information about the type of data you can retain, see security data types.

Another way is to send the logs to a storage account.

Alerts

Security alerts are notifications that are generated when anomalous activity is detected on the resources used by the workload or the platform.

With the Microsoft Defender plans, Microsoft Defender for Cloud analyzes log data and shows a list of alerts that's based on logs collected from resources within a scope. Alerts include context information such as severity, status, activity time. Defender for Cloud also provides a correlated view called **incidents**. Use this data to analyze what actions the attacker took, and what resources were affected. Have strategies to react to alerts as soon as they are generated. An option is to handle alerts in Azure Functions.

Use the data to support these activities:

- Remediation of threats.
- Investigation of an incident.
- Proactive hunting activities.

For more information, see Security alerts and incidents.

Centralize logs and alerts

Organizations typically follow one of three models when deploying logs: centralized, decentralized, or hybrid. The choice depends on organizational structures. For example, if each team owns their resource group, log data is segregated per resource. While access control to that data might be easy to set up, it's difficult to correlate logs. This might be challenging for the SecOps team who need a holistic view to analyze the data.

Consider a central view of log and data, when applicable. Some advantages include:

- The resources in the workload can share a common log workspace reducing duplication.
- Single point of observability with all log data makes it easier consume data for hunting activities, querying, and statistical evaluation.
- The integrated data can be fed into modern machine learning analytics platforms support ingestion of large amounts of information and can analyze large datasets quickly. In addition, these solutions can be tuned to significantly reduce the false positive alerts.

You can collect logs and alerts from various sources centrally in a Log Analytics Workspace, storage account, and Event Hubs. You can then review and query log data efficiently. In Azure Monitor, use the **diagnostic setting** on resources to route specific logs that are important for the organization. Logs vary by resource type. In Microsoft Defender for Cloud, take advantage of the continuous export feature to route alerts.

> **NOTE**
> Platform logs are not available indefinitely. You'll need to keep them so that you can review them later for auditing purposes or offline analysis. Use Azure Storage Accounts for long-term/archival storage. In Azure Monitor, specify a retention period when you enable diagnostic setting for your resources.

Another way to see all data in a single view is to integrate logs and alerts into Security Information and Event Management (SIEM) solutions, such as Microsoft Sentinel. Other popular third-party choices are Splunk, QRadar, ArcSight. Microsoft Defender for Cloud and Azure Monitor supports all of those solutions.

Integrating more data can enrich alerts with additional context. However, collection is not detection. Make sure a high volume of low value data doesn't flow into those solutions.

If you don't have a reasonable expectation that the data will provide value, deprioritize integration of these events. For example, high volume of firewall denies events may create noise without actual actions.

That choice will help in rapid response and remediation by filtering out false positives, and elevate true positives, and so on. Also it will lower SIEM cost, false positives, and increase performance.

Other ways of log integration may involve a hybrid model that mixes centralized and decentralized (distributed among teams) approaches. For details, see Important considerations for an access control strategy.

Next

Responding to alerts is an essential way to prevent security assurance decay, and designing for defense-in depth and least privilege strategies.

Remediate security risks

Related links

For more information, see these articles:

- How to get started with Azure Monitor and third-party SIEM integration
- How to collect platform logs and metrics with Azure Monitor
- Export alerts
- Understand Microsoft Defender for Cloud data collection

Go back to the main article: Monitor

Remediate security risks in Microsoft Defender for Cloud

11/19/2021 • 4 minutes to read • Edit Online

Security controls must remain effective against attackers who continuously improve their ways to attack the digital assets of an enterprise. Use the principle of drive continuous improvement to make sure systems are regularly evaluated and improved.

Start by remediating common security risks. These risks are usually from well-established attack vectors. This will forces attackers to acquire use advanced and more expensive attack methods.

Key points

- Processes for handling incidents and post-incident activities, such as lessons learned and evidence retention.
- Remediate the common risks identified by Microsoft Defender for Cloud.
- Track remediation progress with secure score and comparing against historical results.
- Address alerts and take action with remediation steps.

Track Secure Score

Do you review and remediate common risks in the workload boundary?

Monitor the security posture of VMs, networks, storage, data services, and various other contributing factors. Secure Score in Microsoft Defender for Cloud shows a composite score that represents the security posture at the subscription level.

Do you have a process for formally reviewing Secure Score on Microsoft Defender for Cloud?

As you review the results and apply recommendations, track the progress and prioritize ongoing investments. Higher score indicates a better security posture.

- Set up a regular cadence (typically monthly) to review the secure score and plan initiatives with specific improvement goals.

- Assign stakeholders for monitoring and improving the score. Gamify the activity if possible to increase engagement and focus from the responsible teams.

As a technical workload owner, work with your organization's dedicated team that monitors Secure Score. In the DevOps model, workload teams may be responsible for their own resources. Typically, these teams are responsible.

- Security posture management team
- Vulnerability management or governance, risk, and compliance team
- Architecture team
- Resource-specific technical teams responsible for improving secure score, as shown in this table.

CATEGORY	RESOURCES	RESPONSIBLE TEAM
Compute and applications	App Services	Application Development/Security Team(s)
	Containers	Application Development and/or Infrastructure/IT Operations
	Virtual machines, scale sets, compute	IT/Infrastructure Operations
Data and Storage	SQL/Redis/Data Lake Analytics/Data Lake Store	Database Team
	Storage Accounts	Storage/Infrastructure Team
Identity and access management	Subscriptions	Identity Team(s)
	Key Vault	Information/Data Security Team
Networking Resources		Networking Team and Network Security Team
IoT Security	IoT Resources	IoT Operations Team

The Azure Secure Score sample shows how to get your Azure Secure Score for a subscription by calling the Microsoft Defender for Cloud REST API. The API methods provide the flexibility to query the data and build your own reporting mechanism of your secure scores over time.

Review and remediate recommendations

Microsoft Defender for Cloud monitors the security status of machines, networks, storage and data services, and applications to discover potential security issues. Enable this capability at no additional cost to detect vulnerable virtual machines connected to internet, missing security updates, missing endpoint protection or encryption, deviations from baseline security configurations, missing Web Application Firewall (WAF), and more.

View the recommendations to see the potential security issues and apply the Microsoft Defender for Cloud recommendations to execute technical remediations.

The recommendations are grouped by controls. Each recommendation has detailed information such as severity, affected resources, and quick fixes where applicable. Start with high severity items.

Defender for Cloud has the capability of exporting results at configured intervals. Compare the results with previous sets to verify that issues have been remediated.

For more information, see Continuous export.

Policy remediation

A common approach for maintaining the security posture is through Azure Policy.

Along with organizational policies, a workload owner can use scoped policies for governance purposes, such as check misconfiguration, prohibit certain resource types, and others. The resources are evaluated against rules to identify unhealthy resources that are risky. Post evaluation, certain actions are required as remediation. The actions can be enforced through Azure Policy effects.

For example, a workload runs in an Azure Kubernetes Service (AKS) cluster. The business goals require the workload to run in a highly restrictive environment. As a workload owner, you want the resource group to contain AKS clusters that are private. You can enforce that requirement with the **Deny** effect. It will prevent a cluster from being created if that rule isn't satisfied.

That sort of isolation can be maintained through policies at a higher level such as the subscription level or even management groups.

Another use case is that it can be automatically remediated by deploying related resources. For example, the organization wants all storage resources in a subscription to send logs to a common Log Analytics workspace. If a storage account doesn't pass the policy, a deployment is automatically started as remediation. That remediation can be enforced through **DeployIfNotExist**. There are some considerations.

- There's a significant wait before the resource is updated and the deployment starts. In the preceding example, there won't be logs captured during that wait time. Avoid using this effect for resources that cannot tolerate a delay.
- The resource deployed because of **DeployIfNotExist** are created by a separate identity than that of the identity that did the original deployment. That identity must have high enough privileges to make the required changes.

Manage alerts

Microsoft Defender for Cloud shows a list of alerts that's based on logs collected from resources within a scope. Alerts include context information such as severity, status, activity time. Most alerts have MITRE ATT&CK® tactics that can help you understand the kill chain intent. Select the alert and investigate the problem with detailed information.

Finally take action. That action can be to fix the resources that are out of compliance with actionable remediation steps. You can also suppress alerts that are false positives.

Make sure that you are integrating critical security alerts into Security Information and Event Management (SIEM), Security Orchestration Automated Response (SOAR) without introducing a high volume of low value data. Microsoft Defender for Cloud can stream alerts to Microsoft Sentinel. You can also use a third-party solution by using Microsoft Graph Security API.

Next

Azure security operations

Related links

> Go back to the main article: Monitor

Security audits

11/19/2021 • 5 minutes to read • Edit Online

To make sure that the security posture doesn't degrade over time, have regular auditing that checks compliance with organizational standards. Enable, acquire, and store audit logs for Azure services.

Key points

- Improve secure score in Microsoft Defender for Cloud.
- Use an industry standard benchmark to evaluate your organizations current security posture.
- Perform regular internal and external compliance audits, including regulatory compliance attestations.
- Review the policy requirements.
- Use Azure Governance Visualizer for a holistic overview of your technical Azure Governance implementation.

Evaluate using standard benchmarks

Do you evaluate the security posture of this workload using standard benchmarks?

Use an industry standard benchmark to evaluate your organizations current security posture.

Benchmarking allows you to improve your security program by learning from external organizations. It lets you know how your current security state compares to that of other organizations, providing both external validation for successful elements of your current system and identifying gaps that serve as opportunities to enrich your team's overall security strategy. Even if your security program isn't tied to a specific benchmark or regulatory standard, you will benefit from understanding the documented ideal states by those outside and inside of your industry.

As an example, the Center for Internet Security (CIS) has created security benchmarks for Azure that map to the CIS Control Framework. Another reference example is the MITRE ATT&CK™ framework that defines the various adversary tactics and techniques based on real-world observations. These external references control mappings and help you to understand any gaps between your current strategy, what you have, and what other experts have in the industry.

Suggested action

Develop an Azure security benchmarking strategy aligned to industry standards.

As people in the organization and on the project change, it is crucial to make sure that only the right people have access to the application infrastructure. Auditing and reviewing access control reduces the attack vector to the application. Azure control plane depends on Azure AD and access reviews are often centrally performed as part of internal, or external audit activities.

Make sure that the security team is auditing the environment to report on compliance with the security policy of the organization. Security teams may also enforce compliance with these policies.

Audit regulatory compliance

Compliance is important for several reasons. Aside from signifying levels of standards, like `ISO 27001` and others, noncompliance with regulatory guidelines may bring sanctions and penalties. Regularly review roles that have high privileges. Set up a recurring review pattern to ensure that accounts are removed from permissions as roles change. Consider auditing at least twice a year.

Suggested action

Use Microsoft Defender for Cloud to continuously assess and monitor your compliance score.

Learn more

Assess your regulatory compliance

Have you established a monitoring and assessment solution for compliance?

Continuously assess and monitor the compliance status of your workload. Microsoft Defender for Cloud provides a regulatory compliance dashboard that shows the current security state of workload against controls mandated by the standard governments or industry organizations and Azure Security Benchmark. Keep your resources in compliance with those standards. Defender for Cloud tracks many standards. You can set the standards by management groups in a subscription.

Consider using Azure Access Reviews or Entitlement Management to periodically review access to the workload.

Consider using Azure Access Reviews or Entitlement Management to periodically review access to the workload.

For Azure, use Azure Policy to create and manage policies that enforce compliance. Azure Policies are built on the Azure Resource Manager capabilities. Azure Policy can also be assigned through Azure Blueprints.

For more information, see Tutorial: Create and manage policies to enforce compliance.

Here's an example management group that is tracking compliance to the Payment Card Industry (PCI) standard.

Do you have internal and external audits for this workload?

A workload should be audited internally, external, or both with the goal of discovering security gaps. Make sure that the gaps are addressed through updates.

Auditing is important for workloads that follow a standard. Aside from signifying levels of standards, noncompliance with regulatory guidelines may bring sanctions and penalties.

Perform regulatory compliance attestation. Attestations are done by an independent party that examines if the workload is in compliance with a standard.

Review critical access

Is access to the control plane and data plane of the application periodically reviewed?

Regularly review roles that have high privileges. Set up a recurring review pattern to ensure that accounts are removed from permissions as roles change. Consider auditing at least twice a year.

As people in the organization and on the project change, make sure that only the right people, have access to the application infrastructure and just enough privileges to complete the task. Auditing and reviewing the access control reduces the attack vector to the application.

Azure control plane depends on Azure AD. You can conduct the review manually or through an automated process by using tools such as Azure AD access reviews. These reviews are often centrally performed often as part of internal or external audit activities.

Check policy compliance

Make sure that the security team is auditing the environment to report on compliance with the security policy of the organization. Security teams may also enforce compliance with these policies.

Enforce and audit industry, government, and internal corporate security policies. Policy monitoring checks that initial configurations are correct and that it continues to be compliant over time.

For Azure, use Azure Policy to create and manage policies that enforce compliance. Azure Policies are built on the Azure Resource Manager capabilities. Azure Policy can also be assigned through Azure Blueprints. For more information, see Tutorial: Create and manage policies to enforce compliance.

Capture critical data

Azure Governance Visualizer captures data from the most relevant Azure governance capabilities such as Azure Policy, Azure role-based access control (Azure RBAC), and Azure Blueprints. The visualizer PowerShell script iterates through an Azure tenant's management group hierarchy down to the subscription level. From the collected data, the visualizer shows your hierarchy map, creates a tenant summary, and builds granular scope insights about your management groups and subscriptions.

Next steps

Remediate security risks in Microsoft Defender for Cloud

Related links

- Secure score in Microsoft Defender for Cloud allows you view all the security vulnerabilities into a single score.
- Tutorial: Improve your regulatory compliance describes a step-by-step process to evaluate regulatory requirements in Microsoft Defender for Cloud.

Azure security test practices

11/19/2021 • 3 minutes to read • Edit Online

Regularly test your security design and implementation, as part the organization's operations. That integration will make sure the security assurances are effective and maintained as per the security standards set by the organization.

A well-architected workload should be resilient to attacks. It should recover rapidly from disruption and yet provide the security assurances of confidentiality, integrity, and availability. Invest in simulated attacks as tests that can indicate gaps. Based on the results of the results you can harden the defense and limit a real attacker's lateral movement within your environment.

Simulated tests can also give you data to plan risk mitigation. Applications that are already in production should use data from real-world attacks. New or updated applications with new features, should rely on structured models for detecting risks early, such as threat modeling.

Key points

- Define test cases that are realistic and based on real-world attacks.
- Identify and catalog lowest cost methods for preventing and detecting attacks.
- Use penetration testing as a one-time attack to validate security defenses.
- Simulate attacks through red teams for long-term persistent attacks.
- Measure and reduce the potential attack surface that attackers target for exploitation for resources within the environment.
- Ensure proper follow-up to educate users about the various means that an attacker may use.

Penetration testing (pentesting)

Do you perform penetration testing on the workload?

It's recommended that you simulate a one-time attack to detect vulnerabilities. Pentesting is a popular methodology to validate the security defense of a system. The practitioners are security experts who are not part of the organization's IT or application teams. So, they look at the system in a way that malicious actors scope an attack surface. The goal is to find security gaps by gathering information, analyzing vulnerabilities, and reporting.

Penetration tests provide a point-in-time validation of security defenses. Red teams can help provide ongoing visibility and assurance that your defenses work as designed, potentially testing across different levels within your workload(s). Red team programs can be used to simulate either one time, or persistent threats against an organization to validate defenses that have been put in place to protect organizational resources.

Microsoft recommends penetration testing and red team exercises to validate security defenses for your workload.

Penetration Testing Execution Standard (PTES) provides guidelines about common scenarios and the activities required to establish a baseline.

Azure uses shared infrastructure to host your assets and assets belonging to other customers. In a pentesting exercise, the practitioners may need access to sensitive data of the entire organization. Follow the rules of engagement to make sure that access and the intent is not misused. For guidance about planning and executing simulated attacks, see Penetration Testing Rules of Engagement.

Learn more

- Azure Penetration Testing
- Penetration Testing

Simulate attacks

The way users interact with a system is critical in planning your defense. The risks are even higher for critical impact accounts because they have elevated permissions and can cause more damage.

Do you carry out simulated attacks on users of this workload?

Simulate a persistent threat actor targeting your environment through a red team. Here are some advantages:

- Periodic checks. The workload will get checked through a realistic attack to make sure the defense is up to date and effective.
- Educational purposes. Based on the learnings, upgrade the knowledge and skill level. This will help the users understand the various means that an attacker may use to compromise accounts.

A popular choice to simulate realistic attack scenarios is Office 365 Attack Simulator.

Is personal information detected and removed/obfuscated automatically?

Be cautious about using sensitive application information. Don't store personal information such as contact information, payment information, and so on, in any application logs. Apply protective measures, such as obfuscation. Machine learning tools can help with this measure. For more information, see PII Detection cognitive skill.

Related links

Threat modeling is a structured process to identify the possible attack vectors. Based on the results, prioritize the risk mitigate efforts. For more information, see Application threat analysis.

For more information on current attacks, see the Microsoft Security Intelligence (SIR) report.

Microsoft Cloud Red Teaming

> Go back to the main article: Monitor

Security operations in Azure

11/19/2021 • 4 minutes to read • Edit Online

The responsibility of the security operation team (also known as Security Operations Center (SOC), or SecOps) is to rapidly detect, prioritize, and triage potential attacks. These operations help eliminate false positives and focus on real attacks, reducing the mean time to remediate real incidents. Central SecOps team monitors security-related telemetry data and investigates security breaches. It's important that any communication, investigation, and hunting activities are aligned with the application team.

Here are some general best practices for conducting security operations:

- Follow the NIST Cybersecurity Framework functions as part of operations.
  - **Detect** the presence of adversaries in the system.
  - **Respond** by quickly investigating whether it's an actual attack or a false alarm.
  - **Recover** and restore the confidentiality, integrity, and availability of the workload during and after an attack.

    For information about the framework, see NIST Cybersecurity Framework.
- Acknowledge an alert quickly. A detected adversary must not be ignored while defenders are triaging false positives.
- Reduce the time to remediate a detected adversary. Reduce their opportunity time to conduct and attack and reach sensitive systems.
- Prioritize security investments into systems that have high intrinsic value. For example, administrator accounts.
- Proactively hunt for adversaries as your system matures. This effort will reduce the time that a higher skilled adversary can operate in the environment. For example, skilled enough to evade reactive alerts.

For information about the metrics that the Microsoft's SOC team uses , see Microsoft SOC.

Tools

Here are some Azure tools that a SOC team can use investigate and remediate incidents.

TOOL	PURPOSE
Microsoft Sentinel	Centralized Security Information and Event Management (SIEM) to get enterprise-wide visibility into logs.
Microsoft Defender for Cloud	Alert generation. Use security playbook in response to an alert.
Azure Monitor	Event logs from application and Azure services.
Azure Network Security Group (NSG)	Visibility into network activities.
Azure Information Protection	Secure email, documents, and sensitive data that you share outside your company.

Investigation practices should use native tools with deep knowledge of the asset type such as an Endpoint detection and response (EDR) solution, Identity tools, and Microsoft Sentinel.

For more information about monitoring tools, see Security monitoring tools in Azure.

Assign incident notification contact

Security alerts need to reach the right people in your organization. Establish a designated point of contact to receive Azure incident notifications from Microsoft, and, or Azure Defender for Cloud. In most cases, such notifications indicate that your resource is compromised or attacking another customer. This enables your security operations team to rapidly respond to potential security risks and remediate them.

This enables your security operations team to rapidly respond to potential security risks and remediate them.

Ensure administrator contact information in the Azure enrollment portal includes contact information that will notify security operations directly or rapidly through an internal process.

Learn more

To learn more about establishing a designated point of contact to receive Azure incident notifications from Microsoft, reference the following articles:

- Update notification settings
- Configure email notifications for security alerts

Incident response

Is the organization effectively monitoring security posture across workloads, with a central SecOps team monitoring security-related telemetry data and investigating possible security breaches? Communication, investigation, and hunting activities need to be aligned with the application team(s).

Are operational processes for incident response defined and tested?

Actions executed during an incident and response investigation could impact application availability or performance. Define these processes and align them with the responsible (and in most cases central) SecOps team. The impact of such an investigation on the application has to be analyzed.

Are there tools to help incident responders quickly understand the application and components to do an investigation?

Incident responders are part of a central SecOps team and need to understand security insights of an

application. Security playbook in Microsoft Sentinel can help to understand the security concepts and cover the typical investigation activities.

Suggested action

Consider using Microsoft Defender for Cloud to monitor security-related events and get alerted automatically.

Learn more

Security alerts and incidents in Microsoft Defender for Cloud

Hybrid enterprise view

Security operations tooling and processes should be designed for attacks on cloud and on-premises assets. Attackers don't restrict their actions to a particular environment when targeting an organization. They attack resources on any platform using any method available. They can pivot between cloud and on-premises resources using identity or other means. This enterprise-wide view will enable SecOps to rapidly detect, respond, and recover from attacks, reducing organizational risk.

Leverage native detections and controls

Use Azure security detections and controls instead of creating custom features for viewing and analyzing event logs. Azure services are updated with new features and have the ability to detect false positive with a higher accuracy rate.

Integrating logs from the network devices, and even raw network traffic itself, will provide greater visibility into potential security threats flowing over the wire.

To get a unified view across the enterprise, feed the logs collected through native detections (such as Azure Monitor) into a centralized security information and event management (SIEM) solution like Microsoft Sentinel. Avoid using generalized log analysis tools and queries. Within Azure Monitor, create Log Analytics Workspace to store logs. You can also review logs and perform queries on log data. These tools can offer high-quality alerts.

The modern machine learning-based analytics platforms support ingestion of extremely large amounts of information and can analyze large datasets very quickly. In addition, these solutions can be tuned to significantly reduce false positive alerts.

Examples of network logs that provide visibility include:

- Security group logs - flow logs and diagnostic logs
- Web application firewall logs
- Virtual network taps and their equivalents
- Azure Network Watcher

Suggested actions

Integrate network device log information in advanced SIEM solutions or other analytics platforms.

Learn more

Enable enhanced network visibility

Next steps

- Security health modeling
- Security tools
- Security logs and audits
- Check for identity, network, data risks

Tradeoffs for security

11/19/2021 • 4 minutes to read • Edit Online

Security provides confidentiality, integrity, and availability assurances of an organization's data and systems. When designing a system you can almost never compromise on security controls. When you enhance security of an architecture there might be impact on reliability, performance efficiency, cost, and operational excellence. This article describes some of those considerations.

Security vs Reliability

Reliable applications are resilient and highly available. Every architectural component factors in achieving your requirements for reliability. Workload security is often woven into many layers of the workload's architecture, operations, and runtime requirements; and may come with their own implications on resiliency or availability.

For example, identity providers and authorization services are critical dependencies to consider. This includes the identity service (Microsoft Identity Platform) and any libraries that help facilitate the use of those services. At some points in the architecture, a failure at an identity layer is terminal. At other points, reliability can be still achieved through strategies such as caching, taking advantage of TTLs on access tokens, and others. OAuth2 claims validation can happen mostly disconnected from the claims provider. However, not all authorization can be achieved that way. In those situations reliability may be traded in favor of complete security.

Many workloads may quickly degrade in functionality with the loss of critical security controls. Consider evaluating at each component of your architecture to detect that condition.

Other security considerations that might impact reliability are:

- Poor or manual certifications or key rotation practices. Failure to do those tasks can lead to reliability issues.
- Expired service principals. For example, a deployment pipeline that used a service principal might fail at a later date, if that principal's access key has expired. Using managed identities helps keep reliability high while also maintaining least privileges on that identity.
- High availability is often achieved by redundancy (actively or passively), and security controls also need to align with the failover mechanism. For example, failing over from one storage account to another for reliability may impact how the client's active authorization session is handled. Using managed identity with Azure AD integration for storage access can result in a higher reliability because the client doesn't have to manage SAS tokens when switching to the new storage account.

Security vs Cost Optimization

Increasing security of the workload will almost always lead to higher cost. There are some ways to optimize cost.

- Maximum security may not always be practical for all environments. Evaluate the security requirements in pre-production and production environments. Are services such as Azure DDoS Protection, Microsoft Sentinel, Dedicated HSMs, Microsoft Defender for Cloud needed in pre-production? Is inner loop mocking of security controls sufficient? If resources are not publicly accessible, can you dial down some controls for cost savings? Always make those choices, *if and only if*, the lowered environment still meets the business requirements.

- Premium security features can increase the cost. There are areas you can reduce cost by using native security features. For example, avoid implementing custom roles if you can use built-in roles.

- Every security control has an opportunity to impact workflows, and workflows that involve people can be expensive. A security control that stops work from being done should be evaluated as necessary or

unnecessarily redundant. Total cost of ownership (TCO) includes operational costs for developers, operators, IT SecOps and onerous security protocols. Reach agreement about where "less" can br "sufficient" to optimize costs.

- TCO includes the time needed do tasks. Optimizing that time will optimize cost. Using platform features can lower TCO and enhance the security posture. Instead of training an engineer to manually review logs and correlate access patterns, use intelligence in services, such as Microsoft Defender for Cloud or Sentinel alerts.

Security vs Operational Excellence

Operational Excellence involves understanding business and workload behavior, and applying appropriate amounts of automation, and observability into those processes.

- Release management

  Your organization defines its quality gates (manual or automated) as part of its safe deployment practices. Evaluate whether each gate is required or optional, taking into consideration whether it's valuable to perform, at the added cost of complexity (and time) for that release. Adding security checks into the process makes the checks more valuable at that point in the process than any gains (usually simplicity and time) by not having them in place.

- Organizational policies

  Teams can often benefit from collaborating and using cross-functional skills in all stages of workload life cycle, from development all the way through production. However, organizational policy or regulatory concerns may prevent such wide-reaching access. Consider, where possible, isolating those systems that do need heightened access policies from those that do not. Avoid applying "one size fits all" model to all components in the system. It's easier to optimize operations in systems that allow more unregulated access. For systems that demand regulated access, complexity is expected to increase leading to higher cost.

- Supportability considerations

  The most "serviceable" architectures are the ones that are the most transparent to everyone involved, and those often have the least number of security controls. Adding security controls to your architecture like filtered telemetry feeds, redacted logs, runtime system access restrictions, and so on can all impact the supportability of a solution. Adding security controls often require adding compensating or compromised solutions for observability into the platform.

Related link

Go back to the main article: Security

Overview of the cost optimization pillar

11/19/2021 • 2 minutes to read • Edit Online

The cost optimization pillar provides principles for balancing business goals with budget justification to create a cost-effective workload while avoiding capital-intensive solutions. Cost optimization is about looking at ways to reduce unnecessary expenses and improve operational efficiencies.

Use the pay-as-you-go strategy for your architecture, and invest in scaling out, rather than delivering a large investment-first version. Consider opportunity costs in your architecture, and the balance between first mover advantage versus *fast follow*. Use the cost calculators to estimate the initial cost and operational costs. Finally, establish policies, budgets, and controls that set cost limits for your solution.

To assess your workload using the tenets found in the Microsoft Azure Well-Architected Framework, reference the Microsoft Azure Well-Architected Review.

We recommend exploring the following videos to dive deeper into Azure cost optimization:

Topics

The Microsoft Azure Well-Architected Framework includes the following topics in the cost optimization pillar:

COST TOPIC	DESCRIPTION
Capture cost requirements	Start your planning with a careful enumeration of requirements. Make sure the needs of the stakeholders are addressed. For strong alignment with business goals, those areas must be defined by the stakeholders and shouldn't be collected from a vendor.
Cost of resources in Azure regions	Cost of an Azure service can vary between locations based on demand and local infrastructure costs.
Governance	Understand how governance can assist with cost management. This work will benefit your ongoing cost review process and will offer a level of protection for new resources.
Estimate the initial cost	It's difficult to attribute costs before deploying a workload to the cloud. If you use methods for on-premises estimation or directly map on-premises assets to cloud resources, estimate will be inaccurate.
PaaS	Look for areas in the architecture where it may be natural to incorporate platform-as-a-service (PaaS) options. These options include caching, queues, and data storage. PaaS reduces time and cost of managing servers, storage, networking, and other application infrastructure.

COST TOPIC	DESCRIPTION
Consumption	A common way to estimate cost is by considering workloads on a peak throughput. Under consistently high usage, consumption-based pricing can be less efficient for estimating baseline costs when compared to the equivalent provisioned pricing.
Provision cloud resources	Deployment of workload cloud resources can optimize cost.
Monitor cost	Azure Cost Management has an alert feature. Alerts are generated when consumption reaches a threshold.
Optimize cost	Monitor and optimize the workload by using the right resources and sizes.
Tradeoffs for costs	As you design the workload, consider tradeoffs between cost optimization and other aspects of the design, such as security, scalability, resilience, and operability.

Next section

Read the cost optimization principles to guide you in your overall strategy.

Principles

Checklist - Design for cost

11/19/2021 • 2 minutes to read • Edit Online

Use this checklist when designing a cost-effective workload.

Cost model

- **Capture clear requirements**. Gather detailed information about the business workflow, regulatory, security, and availability.
  - Capture requirements
- **Estimate the initial cost**. Use tools such as Azure pricing calculator to assess cost of the services you plan to use in the workload. Use Azure Migrate and Microsoft Azure Total Cost of Ownership (TCO) Calculator for migration projects. Accurately reflect the cost associated with right storage type. Add hidden costs, such as networking cost for large data download.
  - Estimate the initial cost
- **Define policies for the cost constraints defined by the organization**. Understand the constraints and define acceptable boundaries for quality pillars of scale, availability, security.
  - Consider the cost constraints
- **Identify shared assets**. Evaluate the business areas where you can use shared resources. Review the billing meters build chargeback reports per consumer to identify metered costs for shared cloud services.
  - Create a structured view of the organization in the cloud
- **Plan a governance strategy**. Plan for cost controls through Azure Policy. Use resource tags so that custom cost report can be created. Define budgets and alerts to send notifications when certain thresholds are reached.
  - Governance

Architecture

- **Check the cost of resources in various Azure geographic regions**. Check your egress and ingress cost, within regions and across regions. Only deploy to multiple regions if your service levels require it for either availability or geo-distribution.
  - Azure regions
- **Choose a subscription that is appropriate for the workload**. Azure Dev/Test subscription types are suitable for experimental or non-production workloads and have lower prices on some Azure services such as specific VM sizes. If you can commit to one or three years, consider subscriptions and offer types that support Azure Reservations.
  - Subscription and offer type
- **Choose the right resources to handle the performance**. Understand the usage meters and the number of meters for each resource in the workload. Consider tradeoffs over time. For example, cheaper virtual machines may initially indicate a lower cost but can be more expensive over time to maintain a certain performance level. Be clear about the billing model of third-party services.
  - Azure resources
  - Use cost alerts to monitor usage and spending

- **Compare consumption-based pricing with pre-provisioned cost**. Establish baseline cost by considering the peaks and the frequency of peaks when analyzing performance.
  - Consumption and fixed cost models
- **Use proof-of-concept deployments**. The Azure Architecture Center has many reference architectures and implementations that can serve as a starting point. The Azure Tech Community has architecture and services forums.
- **Choose managed services when possible**. With PaaS and SaaS options, the cost of running and maintaining the infrastructure is included in the service price.
  - Managed services

Develop a cost model

11/19/2021 • 6 minutes to read • Edit Online

Cost modeling is an exercise where you create logical groups of cloud resources that are mapped to the organization's hierarchy and then estimate costs for those groups. The goal of cost modeling is to estimate the overall cost of the organization in the cloud.

1. **Understand how your responsibilities align with your organization**

   Map the organization's needs to logical groupings offered by cloud services. This way the business leaders of the company get a clear view of the cloud services and how they're controlled.

2. **Capture clear requirements**

   Start your planning with a careful enumeration of requirements. From the high-level requirements, narrow down each requirement before starting on the design of the solution.

3. **Consider the cost constraints**

   Evaluate the budget constraints on each business unit and determine the governance policies in Azure to lower cost by reducing wastage, overprovisioning, or expensive provisioning of resources.

4. **Consider tradeoffs**

   Optimal design doesn't equate to a lowest-cost design.

   As requirements are prioritized, cost can be adjusted. Expect a series of tradeoffs in the areas that you want to optimize, such as security, scalability, resilience, and operability. If the cost to address the challenges in those areas is high, stakeholders will look for alternate options to reduce cost. There might be risky choices made in favor or a cheaper solution.

5. **Derive functional requirements from high-level goals**

   Break down the high-level goals into functional requirements for the components of the solution. Each requirement must be based on realistic metrics to estimate the actual cost of the workload.

6. **Consider the billing model for Azure resources**

   Azure services are offered with consumption-based price where you're charged for only what you use. There's also options for fixed price where you're charged for provisioned resources

   Most services are priced based on units of size, amount of data, or operations. Understand the meters that are used to track usage. For more information, see Azure resources.

At the end of this exercise, you should have identified the lower and upper limits on cost and set budgets for the workload. Azure allows you to create and manage budgets in Azure Cost Management. For information, see Quickstart: Create a budget with an Azure Resource Manager template.

HAVE FREQUENT AND CLEAR COMMUNICATION WITH THE STAKEHOLDERS

In the initial stages, communication between stakeholders is vital. The overall team must align on the requirements so that overall business goals are met. If not, the entire solution might be at risk.

For instance, the development team indicates that the resilience of a monthly batch-processing job is low. They might request the job to work as a single node without scaling capabilities. This request opposes the architect's recommendation to automatically scale out, and route requests to worker nodes.

This type of disagreement can introduce a point of failure into the system, risking the Service Level Agreement, and cause an increase in operational cost.

Organization structure

Map the organization's needs to logical groupings offered by cloud services. This way the business leaders of the company get a clear view of the cloud services and how they're controlled.

1. Understand how your workload fits into cost optimization across the portfolio of cloud workloads.

   If you are working on a workload that fits into a broader portfolio of workloads, see CAF to get started guide to document foundational decisions. That guide will help your team capture the broader portfolio view of business units, resources organizations, responsibilities, and a view of the long term portfolio.

   If cost optimization is being executed by a central team across the portfolio, see CAF to get started managing enterprise costs.

2. Encourage a culture of democratized cost optimization decisions.

   As a workload owner, you can have a measurable impact on cost optimization. There are other roles in the organization which can help improve cost management activities. To help embed the pillar of cost optimization into your organization beyond your workload team, see the CAF article: Build a cost-conscious organization.

3. Reduce costs through shared cloud services and landing zones.

   If your workload has dependencies on shared assets like Active Directory, Network connectivity, security devices, or other services that are also used by other workloads, encourage your central IT organization to provide those services through a centrally managed landing zone to reduce duplicate costs. See the CAF article: get started with centralized design & configuration to get started with the development of landing zones.

4. Calculate the ROI by understanding what is included in each grouping and what isn't.

   Which aspects of the hierarchy are covered by cloud services?

   Azure pricing model is based on expenses incurred for the service. Expenses include hardware, software, development, operations, security, and data center space to name a few. Evaluate the cost benefit of shifting away from owned technology infrastructure to leased technology solutions.

5. Identify scenarios where you can use shared cloud services to lower cost.

   Can some services be shared by other consumers?

   Identify areas where a service or an application environment can be shared with other business units.

   Identify resources that can be used as shared services and review their billing meters. Examples include a virtual network and its hybrid connectivity or a shared app service environment (ASE). If the meter data isn't able to be split across consumers, decide on custom solutions to allocate proportional costs. Move

shared services to dedicated resources for consumers for cost reporting.

> Build chargeback reports per consumer to identify metered costs for shared cloud services. Aim for granular reports to understand which workload is consuming what amount of the shared cloud service.

Next step

Capture cost requirements

Cost constraints

Here are some considerations for determining the governance policies that can assist with cost management.

- What are the budget constraints set by the company for each business unit?
- What are policies for the budget alert levels and associated actions?
- Identify acceptable boundaries for scale, redundancy, and performance against cost.
- Assess the limits for security. Don't compromise on security. Premium cloud security features can drive the cost up. It's not necessary to overinvest. Instead use the cost profile to drive a realistic threat profile.
- Identify unrestricted resources. These resources typically need to scale and consume more cost to meet demand.

Next step

Consider tradeoffs

Functional requirements

Break down high-level goals into functional requirements. For each of those requirements, define metrics to calculate cost estimates accurately. Cloud services are priced based on performance, features, and locations. When defining these metrics, identify acceptable boundaries of performance, scale, resilience, and security. Start by expressing your goals in number of business transactions over time, breaking them down to fine-grain requirements.

What resources are needed for a single transaction, and how many transactions are done per second, day, year?

> Start with a fixed cost of operations and a rough estimate of transaction volume to work out a cost-per-transaction to establish a baseline. Consider the difference between cost models based on fixed, static provisioning of services, more variable costs based upon autoscaling such as serverless technologies.

Use T-shirt sizes for choosing SKUs

When choosing options for services, start with an abstract representation of size. For example, if you choose a T-shirt size approach, small, medium, large sizes, can represent an on-demand virtual machine instead of picking specific virtual machines or managed disks SKU sizes.

Abstract sizes give you an idea of the expected performance and cost requirements. It sets the tone for various consumption units that are used to measure compute resources for performance. Also, it helps in understanding the on-demand consumption model of the services.

For more information, see Estimate the initial cost.

Next steps

Estimate the initial cost

Capture cost requirements

11/19/2021 • 4 minutes to read • Edit Online

Start your planning with a careful enumeration of requirements. Make sure the needs of the stakeholders are addressed. For strong alignment with business goals, those areas must be defined by the stakeholders and shouldn't be collected from a vendor.

Capture requirements at these levels:

- Business workflow
- Compliance and regulatory
- Security
- Availability

What do you aim to achieve by building your architecture in the cloud?

Here are some common answers.

- Take advantage of features only available in the cloud, such as intelligent security systems, regions footprint, or resiliency features.
- Use the on-demand nature of the cloud to meet peak or seasonal requirements, then releasing that cost investment when it is no longer needed.
- Consolidate physical systems.
- Retire on-premises infrastructure.
- Reduce hardware or data center management costs.
- Increase performance or processing capabilities, through services like big data and machine learning.
- Meet regulatory considerations, including taking advantage of certified infrastructure.

Narrow down each requirement before you start the design of the workload. Expect the requirements to change over time as the solution is deployed and optimized.

Landing zone

Consider the cost implications of the geographic region to which the landing zone is deployed.

The landing zone consists of the subscription and resource group, in which your cloud infrastructure components exist. This zone impacts the overall cost. Consider the tradeoffs. For example, there are additional costs for network ingress and egress for cross-zonal traffic. For more information, see Azure regions and Azure resources.

For information about landing zone for the entire organization, see CAF: Implement landing zone best practices.

Security

Security is one of the most important aspects of any architecture. Security measures protect the valuable data of the organization. It provides confidentiality, integrity, and availability assurances against attacks and misuse of the systems.

Factor in the cost of security controls, such as authentication, MFA, conditional access, information protection, JIT/PIM, and premium Azure AD features. Those options will drive up the cost.

For security considerations, see the Security Pillar.

Business continuity

Does the application have a Service Level Agreement that it must meet?

Factor in the cost when you create high availability and disaster recovery strategies.

Overall Service Level Agreement (SLA), Recovery Time Objective (RTO), and Recovery Point Objective (RPO) may drive towards expensive design choices in order to support higher availability requirements. For example, a choice might be to host the application across regions, which is costlier than single region but supports high availability.

If your service SLAs, RTOs and RPOs allow, then consider cheaper options. For instance, pre-build automation scripts and packages that would redeploy the disaster recovery components of the solution from the ground-up in case a disaster occurs. Alternatively, use Azure platform-managed replication. Both options can lower cost because fewer cloud services are pre-deployed and managed, reducing wastage.

In general, if the cost of high availability exceeds the cost of application downtime, then you could be over engineering the high availability strategy. Conversely, if the cost of high availability is less than the cost of a reasonable period of downtime, you may need to invest more.

Suppose the downtime costs are relatively low, you can save by using recovery from your backup and disaster recovery processes. If the downtime is likely to cost a significant amount per hour, then invest more in the high availability and disaster recovery of the service. It's a three-way tradeoff between cost of service provision, the availability requirements, and the organization's response to risk.

Application lifespan

Does your service run seasonally or follow long-term patterns?

For long running applications, consider using Azure Reservations if you can commit to one-year or three-year term. VM reservations can reduce cost by 60% or more when compared to pay-as-you-go prices.

Reservation is still an operational expense with all the corresponding benefits. Monitor the cost on workloads that have been running in the cloud for an extended period to forecast the reserved instance sizes that are needed. For information about optimization, see Reserved instances.

If your application runs intermittently, consider using Azure Functions in a consumption plan so you only pay for compute resources you use.

Automation opportunities

Is it a business requirement to have the service be available 24x7?

You may not have a business goal to leave the service running all the time. Doing so will incur a consistent cost. Can you save by shutting down the service or scaling it down outside normal business hours? If you can,

- Azure has a rich set of APIs, SDKs, and automation technology that utilizes DevOps and traditional automation principles. Those technologies ensure that the workload is available at an appropriate level of scale as needed.
- Repurpose some compute and data resources for other tasks that run out of regular business hours. See the Compute Resource Consolidation pattern and consider containers or elastic pools for more compute and data cost flexibility.

Budget for staff education

Keep the technical staff up to date in cloud management skills so that the invested services are optimally used.

- Consider using resources such as FastTrack for Azure and Microsoft Learn to onboard the staff. Those resources provide engineering investments at no cost to customers.
- Identify training requirements and costs for cloud migration projects, application development, and architecture refinement.
- Invest in key areas, such as identity management, security configuration, systems monitoring, and automation.
- Give the staff access to training and relevant announcements. This way, they can be aware of new cloud capabilities and updates.
- Provide opportunities to get real-world experience of customers across the globe through conferences, specific cloud training, and passing dedicated Microsoft Exams (AZ, MS, MB, etc.).

Standardization

Ensure that your cloud environments are integrated into any IT operations processes. Those operations include user or application access provisioning, incident response, and disaster recovery. That mapping may uncover areas where additional cloud cost is needed.

Next step

Determine the cost constraints

Azure regions

11/19/2021 • 2 minutes to read • Edit Online

Cost of an Azure service can vary between locations based on demand and local infrastructure costs. Consider all these geographical areas when choosing the location of your resources to estimate costs.

TERMINOLOGY	DESCRIPTION
Azure region	A set of datacenters deployed within a latency-defined perimeter and connected through a dedicated regional low-latency network.
Availability zone	A unique physical location within a region with independent power, network, and cooling to be tolerant to datacenter failures through redundancy and logical isolation of services.
Billing zone	A geographic collection of regions that is used for billing.
Location	A region or a zone within a region. Azure has datacenters all over the world and each datacenter is placed in a location.
Landing zone	The ultimate location of your cloud solution or the landing zone, typically consisting of logical containers such as a subscription and resource group, in which your cloud infrastructure components exist.

The complete list of Azure geographies, regions, and locations, is shown in Azure global infrastructure.

To see availability of a product by region, see Products available by region.

Tradeoff

- Locating resources in a cheaper region should not negate the cost of network ingress and egress or by degraded application performance because of increased latency.
- An application hosted in a single region may cost less than an application hosted across regions because of replication costs or the need for extra nodes.

Compliance and regulatory

Azure also offers differentiated cloud regions for specific security and compliance requirements.

Does your solution need specific levels of security and compliance?

If your solution needs to follow certain government regulations, the cost will be higher. Otherwise you can meet less rigid compliance, through Azure Policy, which is free.

Certain Azure regions are built specifically for high compliance and security needs. For example, with Azure Government (USA) you're given an isolated instance of Azure. Azure Germany has datacenters that meet privacy certifications. These specialized regions have higher cost.

Regulatory requirements can dictate restrictions on data residency. These requirements may impact your data replication options for resiliency and redundancy.

Traffic across billing zones and regions

Cross-regional traffic and cross-zonal traffic incur additional costs.

Is the application critical enough to have the footprint of the resources cross zones and,or cross regions?

Bandwidth refers to data moving in and out of Azure datacenters. Inbound data transfers (data going into Azure datacenters) are free for most services. For outbound data transfers, (data going out of Azure datacenters) the data transfer pricing is based on billing zones. For more information, see Bandwidth Pricing Details.

Suppose, you want to build a cost-effective solution by provisioning resources in locations that offer the lowest prices. The dependent resources and their users are located in different parts of the world. In this case, data transfer between locations will add cost if there are meters tracking the volume of data moving across locations. Any savings from choosing the cheapest location could be offset by the additional cost of transferring data.

- The cross-regional and cross-zone additional costs do not apply to global services, such as Azure Active Directory.
- Not all Azure services support zones and not all regions in Azure support zones.

> Before choosing a location, consider how important is the application to justify the cost of having resources cross zones and/or cross regions. For non-mission critical applications such as, developer or test, consider keeping the solution and its dependencies in a single region or single zone to leverage the advantages of choosing the lower-cost region.

Azure resources

11/19/2021 • 3 minutes to read • Edit Online

Just like on-premises equipment, there are several elements that affect monthly costs when using Azure services.

Usage meters for resources

Most services are priced based on units of size, amount of data, or operations. When you provision an Azure resource, Azure creates metered instances for that resource. The *meters* track the resources' usage and generate a usage record that is used to calculate your bill.

For example, you provision a virtual machine in Azure. Some meters that track its usage include: Compute Hours, IP Address Hours, Data Transfer In, Data Transfer Out, Standard Managed Disk, Standard Managed Disk Operations, Standard IO-Disk, Standard IO-Block Blob Read, Standard IO-Block Blob Write, Standard IO-Block Blob Delete.

How is the usage tracked for each resource in the workload?

For each Azure resource, have a clear understanding of the meters that track usage and the number of meters associated with the resource tier. The meters correlate to several billable units. Those units are charged to the account for each billing period. The rate per billable unit depends on the resource tier.

A resource tier impacts pricing because each tier offers levels of features such as performance or availability. For example, a Standard HDD hard disk is cheaper than a Premium SSD hard disk.

> Start with a lower resource tier then scale the resource up as needed. Growing a service with little to no downtime is easier when compared to downscaling a service. Downscaling usually requires deprovisioning or downtime. In general, choose scaling out instead of scaling up.

As part of the requirements, consider the metrics for each resource and build your alerts on baseline thresholds for each metric. The alerts can be used to fine-tune the resources. For more information, see Respond to cost alert.

Allocated usage for the resource

Another way to look at pricing is the allocated usage.

Suppose, you de-allocate the virtual machine. You'll not be billed for Compute Hours, I/O reads or writes, and other compute meters because the virtual machine is not running and has no given compute resources. However, you'll be charged for storage costs for the disks.

Here are some considerations:

- The meters and pricing vary per product and often have different pricing tiers based on the location, size, or capacity of the resource.
- Cost for associated with infrastructure is kept low with commodity hardware.
- Failures cannot be prevented but the effects of failure can be minimized through design choices. The resiliency features are factored in the price of a service and its features.

Here are some examples:

Azure Disk

Start with a small size in GB for a managed disk instead of pay-per-GB model. It's cost effective because cost is incurred on the allocated storage.

ExpressRoute

Start with a smaller bandwidth circuit and scale up as needed.

Compute infrastructure

Deploy an additional smaller instance of compute alongside a smaller unit in parallel. That approach is more cost effective in comparison to restarting an instance to scale up.

For details about how billing works, see Azure Cost Management + Billing documentation.

Subscription and offer type

What is the subscription and offer type in which resources are created?

Azure usage rates and billing periods can vary depending on the subscription and offer type. Some subscription types also include usage allowances or lower prices. For example, Azure Dev/Test subscription offers lower prices on Azure services such as specific VM sizes, PaaS web apps, and VM images with pre-installed software. Visual Studio subscribers obtain as part of their benefits access to Azure subscriptions with monthly allowances.

For information about the subscription offers, see Microsoft Azure Offer Details.

As you decide the offer type, consider the types that support Azure Reservations. With reservations, you prepay for reserved capacity at a discount. Reservations are suitable for workloads that have a long-term usage pattern. Combining the offer type with reservations can significantly lower the cost. For information about subscription and offer types that are eligible for reservations, see Discounted subscription and offer types.

Billing structure for services in Azure Marketplace

Are you considering third-party services through Azure Marketplace?

Azure Marketplace offers both the Azure products and services from third-party vendors. Different billing structures apply to each of those categories. The billing structures can range from free, pay-as-you-go, one-time purchase fee, or a managed offering with support and licensing monthly costs.

Governance

11/19/2021 • 2 minutes to read • Edit Online

Governance can assist with cost management. This work will benefit your ongoing cost review process and will offer a level of protection for new resources.

Understand how centralized governance functions can support your team

Centralized governance can relieve some of the burden related to on-going cost monitoring and optimization. However, that is an augmentation of the workload team's responsibilities, not a replacement. For an understanding of how centralized cloud governance teams operate, see the Cloud Adoption Framework's Govern methodology.

- For more detailed information on cost optimization, see the section on Cost Management discipline.
- For an example of the types of guardrails provided by a governance team, see the narrative for improving the cost management discipline, which includes examples of suggested tags and policies for improving cost governance.
- If your team isn't supported by centralized governance teams, see Cloud governance function to better understand the types of activities your team may need to consider including in each sprint.

Follow organizational policies that define cost boundaries

Use policies to ensure compliance to the identified cost boundaries. Also, it eliminates the need for manual resource approval and speeds up the total provisioning time.

Azure Policy can set rules on management groups, subscriptions, and resources groups. The policies control clouds service resource SKU size, replication features, and locations that are allowed. Use policies to prevent provisioning of expensive resources. Identify the built-in Azure policies that can help lower cost. For additional control, create custom policies.

For more information, see Create management groups for resource organization and management.

Control the group who can manage resources in the subscription, see Azure built-in roles.

Set limits or quotas to prevent unexpected costs, For more information, see Azure subscription and service limits, quotas, and constraints.

Enforce resource tagging

Use tags on resources and resource groups to track the incurred costs. Identify the service meters that can't be tagged or viewed in the cost analysis tool in Azure portal.

The advantages of tagging include:

- The cost can be reported to an owner, an application, a business department or a project initiative. This feature is useful because the overall cost can span multiple resources, locations, and subscriptions.
- Filter information. This filtering can be used in cost analysis tool in Azure portal allowing you to get granular reports.

There are some limitations:

- Not all Azure resources can be tagged and not all taggable resources in Azure are accounted for in the Azure cost analysis tool.

Estimate the initial cost

11/19/2021 • 6 minutes to read • Edit Online

It's difficult to attribute costs before deploying a workload to the cloud. If you use methods for on-premises estimation or directly map on-premises assets to cloud resources, estimate will be inaccurate. For example, if you build your own datacenter your costs may appear comparable to cloud. Most on-premises estimates don't account for costs like cooling, electricity, IT and facilities labor, security, and disaster recovery.

Here are some best practices:

- Use proof-of-concept deployments to help refine cost estimates.
- Choose the right resources that can handle the performance of the workload. For example, cheaper virtual machines may initially indicate a lower cost but can be more expensive eventually to maintain a certain performance level.
- Accurately reflect the cost associated with right storage type.
- Add hidden costs, such as networking cost for large data download.

Migration workloads

Quantify the cost of running your business in Azure by calculating Total Cost Ownership (TCO) and the Return on Investment (ROI). Compare those metrics to existing on-premises equivalents.

It's difficult to attribute costs before migrating to the cloud.

Using on-premises calculation may not accurately reflect the cost of cloud resources. Here are some challenges:

- On-premises TCO may not accurately account for hidden expenses. These expenses include under-utilization of purchased hardware or network maintenance costs including labor and equipment failure.
- Cloud TCO may not accurately account for a drop in the organization's operational labor hours. Cloud provider's infrastructure, platform management services, and additional operational efficiencies are included in the cloud service pricing. Especially true at a smaller scale, the cloud provider's services don't result in reduction of IT labor head count.
- ROI may not accurately account for new organizational benefits because of cloud capabilities. It's hard to quantify improved collaboration, reduced time to service customers, and fast scaling with minimal or no downtime.
- ROI may not accurately account for business process re-engineering needed to fully adopt cloud benefits. In some cases, this re-engineering may not occur at all, leaving an organization in a state where new technology is used inefficiently.

Azure provides these tools to determine cost.

- Microsoft Azure Total Cost of Ownership (TCO) Calculator to reflect all costs.

  > For migration projects, the TCO Calculator may assist, as it pre-populates some common cost but allows you to modify the cost assumptions.

- Azure pricing calculator to assess cost of the services you plan to use in your solution.
- Azure Migrate to evaluate your organization's current workloads in on-premises datacenters. It suggests Azure replacement solution, such virtual machine sizes based on your workload. It also provides a cost estimate.

Example estimate for a microservices workload

Let's consider this scenario as an example. We'll use the Azure Pricing calculator to estimate the initial cost before the workload is deployed. The cost is calculated per month or for 730 hours.

In this example, we've chosen the microservices pattern. As the container orchestrator, one of the options could be Azure Kubernetes Service (AKS) that manages a cluster of pods. We choose NGINX ingress controller because it's well-known controller for such workloads.

> The example is based on the current price and is subject to change. The calculation shown is for information purposes only.

Compute

For AKS, there's no charge for cluster management.

For AKS agent nodes, there are many instance sizes and SKUs options. Our example workload is expected to follow a long running pattern and we can commit to three years. So, an instance that is eligible for reserved instances would be a good choice. We can lower the cost by choosing the **3-year reserved plan**.

The workload needs two virtual machines. One is for the backend services and the other for the utility services.

The B12MS instance with 2 virtual machines is sufficient for this initial estimation. We can lower cost by choosing reserved instances.

Estimated Total: $327.17 per month with upfront payment of $11,778.17.

Application gateway

For this scenario, we consider the **Standard_v2 Tier** of Azure Application Gateway because of the autoscaling capabilities and performance benefits. We also choose consumption-based pricing, which is calculated by capacity units (CU). Each capacity unit is calculated based on compute, persistent connections, or throughput. For Standard_v2 SKU - Each compute unit can handle approximately 50 connections per second with RSA 2048-bit key TLS certificate. For this workload, we estimate 10 capacity units.

Estimated Total: $248.64 per month.

Load balancer

NGINX ingress controller deploys a load balancer that routes internet traffic to the ingress. Approximately 15 load balancer rules are needed. NAT rules are free. The main cost driver is the amount of data processed inbound and outbound independent of rules. We estimate traffic of 1 TB (inbound and outbound).

Estimated Total: $96.37 per month.

Bandwidth

We estimate 2-TB outbound traffic. The first 5 GB/month are free in Zone 1 (Zone 1 includes North America, Europe, and Australia). Between 5 GB - 10 TB /month is charged $0.087 per GB.

Estimated Total: $177.74 per month

External data source

Because the schema-on read nature of the data handled by the workload, we choose Azure Cosmos DB as the external data store. By using the Cosmos DB capacity calculator, we can calculate the throughput to reserve.

Cost variables

- For lower latency, in this scenario we enable geo-replication by using the **Multi-regions writes** feature. By default Cosmos DB uses one region for writes and the rest for reads.
- Default choices in consistency model of **Session** and indexing policy to **Automatic**. Automatic indexing makes Cosmos DB to index all properties in all items for flexible and efficient queries. Custom indexing policy allows you to include/exclude properties from the index for lower write RU and storage size. So uploading a custom indexing policy can make you reduce costs.
- Total data store isn't a significant cost driver, here it's set to 500 GB.
- The throughput is variable indicating peaks. The percentage of time at peak is set to 10%.
- The item size is an average of 90k. By using the capacity calculator you can upload sample json files with your document's data structure, the average document's size, the number of reads/writes per second. These variables have the largest impact on cost because they're used to calculate the throughput. The throughput values are shown in the image.

Now, we use those values in the Azure Pricing calculator.

Azure Cosmos DB

Throughput

Multiple Region Write (Multi-Master)

Savings Options

Save up to 65% on pay as you go prices with 1-year or 3-year reserved capacity for Azure Cosmos DB.

- Pay as you go
- 1 year reserved capacity
- 3 year reserved capacity

$1,635.20
Average per month
($58,867.20 charged upfront)

RUs/sec:
20,000
Region:
East US

= $1,635.20
Effective cost per month
$58,867.20 Charged upfront

Reserved Capacity reservations are automatically applied across regions. For more information, read our Reserved Capacity docs.

Storage

500
GB

Region: East US	500 GB	×	$0.250 Per GB/month	= $125.00

Upfront cost	$58,867.20
Monthly cost	$125.00

The average throughput based on these settings is 20,000 RUs/sec which is the minimum throughput required for a **3-year reserved capacity** plan.

Here is the total cost for three years using the reserved plan:

$1,635.20 Average per month ($58,867.20 charged upfront)

You save $700.00 by choosing the 3-year reserved capacity over the pay-as-you-go price.

CI/CD pipelines

With Azure DevOps, **Basic Plan** is included for Visual Studio Enterprise, Professional, Test Professional, and MSDN Platforms subscribers. And no charge for adding or editing work items and bugs, viewing dashboards, backlogs, and Kanban boards for stakeholders.

Basic plan license for five users is free.

Additional services

For Microsoft Hosted Pipelines, the **Free** tier includes one parallel CI/CD job with 1,800 minutes (30 hours) per month. However you can select the **Paid** tier and have one CI/CD parallel job ($40.00), in this tier, each parallel CI/CD job includes unlimited minutes.

> For this stage of cost estimation, the Self Hosted Pipelines is not required because the workload doesn't have custom software that runs in your build process which isn't included in the Microsoft-hosted option.

Azure Artifacts is a service where you can create package feeds to publish and consume Maven, npm, NuGet, Python, and universal packages. Azure Artifacts is billed on a consumption basis, and is free up until 2 GB of storage. For this scenario, we estimate 56 GB in artifacts ($56.00)

Azure DevOps offers a cloud-based solution for load testing your apps. Load tests are measured and billed in virtual user minutes (VUMs). For this scenario, we estimate a 200,000 VUMs ($72.00).

Estimated Total: $168.00 per month

Managed services

11/19/2021 • 2 minutes to read • Edit Online

Look for areas in the architecture where it may be natural to incorporate platform-as-a-service (PaaS) options. These include caching, queues, and data storage. PaaS reduces time and cost of managing servers, storage, networking, and other application infrastructure.

With PaaS, the infrastructure cost is included in the pricing model of the service. For example, you can provision a lower SKU virtual machine as a jumpbox. There are additional costs for storage and managing a separate server. You also need to configure a public IP on the virtual machine, which is not recommended. A managed service such as Azure Bastion takes into consideration all those costs and offers better security.

Azure provides a wide range of PaaS resources. Here are some examples of when you might consider PaaS options:

TASK	USE
Host a web server	Azure App Service instead of setting up IIS servers.
Indexing and querying heterogenous data	Azure Cognitive Search instead of ElasticSearch.
Host a database server	Azure offers many SQL and no-SQL options such as Azure SQL Database and Azure Cosmos DB.
Secure access to virtual machine	Azure Bastion instead of virtual machines as jump boxes.
Network security	Azure Firewall instead of virtual network appliances.

For more information, see Use platform as a service (PaaS) options.

Reference architecture

To see an implementation that provides better security and lowers cost through PaaS services, see Network DMZ between Azure and an on-premises datacenter.

Consumption and fixed cost models

11/19/2021 • 2 minutes to read • Edit Online

The common pricing options for Azure services are:

- Consumption-based price - You are charged for only what you use. This model is also known as the Pay-As-You-Go rate.
- Fixed price - You provision resources and are charged for those instances whether or not they are used.

A common way to estimate cost is by considering workloads on a peak throughput. Under consistently high utilization, consumption-based pricing can be less efficient for estimating baseline costs when compared to the equivalent provisioned pricing. PaaS and serverless technologies can help you understand the economy cutoff point for consumption-based pricing.

Observe the difference between cost models based on fixed, static provisioning of services, more variable costs based on autoscaling of serverless technologies.

Start with a fixed minimum level of performance and then use architectural patterns (such as Queue Based Load Leveling) and autoscaling of services. With this approach the peaks can be smoothed out into a more consistent flow of compute and data. This approach should temporarily extend your burst performance when the service is under sustained load. If cost is an important factor but you need to maintain service availability under burst workload use the Throttling pattern to maintain quality of service under load.

Compare and contrast the options and understand how to provision workloads that can potentially switch between the two models. The model will be a tradeoff between scalability and predictability. Ideally in the architecture, blend the two aspects.

Provisioning cloud resources to optimize cost

11/19/2021 • 2 minutes to read • Edit Online

Deployment of cloud resources of a workload is known as *provisioning*.

Use the Azure Pricing calculator to estimate the cost of your SKU choices. This section describes some considerations. This list is not meant to be an exhaustive list, but a subset of options.

- AI + Machine Learning
- Big data analytics
- Compute
- Networking
- Data stores

AI + Machine Learning cost estimates

11/19/2021 • 3 minutes to read • Edit Online

The main cost driver for machine learning workloads is the compute cost. Those resources are needed to run the training model and host the deployment. For information about choosing a compute target, see What are compute targets in Azure Machine Learning?.

The compute cost depends on the cluster size, node type, and number of nodes. Billing starts while the cluster nodes are starting, running, or shutting down.

With services such as Azure Machine Learning, you have the option of creating fix-sized clusters or use autoscaling.

> If the amount of compute is not known, start with a zero-node cluster. The cluster will scale up when it detects jobs in the queue. A zero-node cluster is not charged.

Fix-sized clusters are appropriate for jobs that run at a constant rate and the amount of compute is known and measured beforehand. The time taken to spin up or down a cluster incurs additional cost.

> If you don't need retraining frequently, turn off the cluster when not in use.

To lower the cost for experimental or development workloads, choose Spot VMs. They aren't recommended for production workloads because they might be evicted by Azure at any time. For more information, see Use Spot VMs in Azure.

For more information about the services that make up a machine learning workload, see What are the machine learning products at Microsoft?.

This article provides cost considerations for some technology choices. This is not meant to be an exhaustive list, but a subset of options.

Azure Machine Learning

Training models don't incur the machine learning service surcharge. You're charged for these factors.

- The cost is driven by compute choices, such as, the virtual machine sizes and the region in which they are available. If you can commit to one or three years, choosing reserved instances can lower cost. For more information, see Reserved instances.
- As part of provisioning Machine Learning resources, additional resource are deployed such as Azure Container Registry, Azure Block Blob Storage, and Key Vault. You're charged for as per the pricing of those individual services.
- If you deploy models to a Kubernetes Service cluster, Machine Learning adds a surcharge on top of the Kubernetes Service compute cost. This cost can be lowered through autoscaling.

For more information, see these articles:

- Machine Learning pricing calculator
- Azure Machine Learning

Reference architecture

- Training of Python scikit-learn and deep learning models on Azure
- Distributed training of deep learning models on Azure
- Batch scoring of Python machine learning models on Azure
- Batch scoring of deep learning models on Azure
- Real-time scoring of Python scikit-learn and deep learning models on Azure
- Machine learning operationalization (MLOps) for Python models using Azure MachineLearning
- Batch scoring of R machine learning models on Azure
- Real-time scoring of R machine learning models on Azure
- Batch scoring of Spark machine learning models on Azure Databricks
- Enterprise-grade conversational bot
- Build a real-time recommendation API on Azure

Azure Cognitive Services

The billing depends on the type of service. The charges are based on the number of transactions for each type of operation specific to a service. Certain number of transactions are free. If you need additional transactions, choose from the **Standard** instances. For more information, see

- Cognitive services pricing calculator
- Cognitive services pricing

Reference architecture

Build an enterprise-grade conversational bot

Azure Bot Service

The Azure Bot Service is a managed service purpose-built for enterprise-grade bot development. Billing is based on the number of messages. Certain number of messages are free. If you need to create custom channels, choose **Premium channels**, which can drive up the cost of the workload.

For a Web App Bot, an Azure App Service is provisioned to host the bot. Also, an instance of Application Insights is provisioned. You're charged for as per the pricing of those individual services.

Reference architecture

Enterprise-grade conversational bot

Big data analytics cost estimates

11/19/2021 • 6 minutes to read • Edit Online

Most big data workloads are designed to do:

- Batch processing of big data sources at rest.
- Stream processing of data in motion.

Those workloads have different needs. Batch processing is done with long-running batch jobs. For stream processing, the data ingestion component should be able to capture and in some cases buffer, store real-time messages. Both workloads also have the requirement to store large volume of data. Then, filter, aggregate, and prepare that data for analysis.

For information about choosing technologies for each workload, see these articles:

- Batch processing
  - Technology choices for batch processing
  - Capability matrix
- Stream processing
  - What are your options when choosing a technology for real-time processing?
  - Capability matrix

This article provides cost considerations for some of those choices. This is not meant to be an exhaustive list, but a subset of options.

Azure Synapse Analytics

The analytics resources are measured in *Data Warehouse Units (DWUs)*, which tracks CPU, memory, and IO. DWU also indicates the required level of performance. If you need higher performance, add more DWU blocks.

You can provision the resources in one of two service levels.

- **Compute Optimized Gen1** tracks usage in DWUs and is offered in a pay-as-you-go model.
- **Compute Optimized Gen2** tracks the compute DWUs (cDWUs) which allows you to scale the compute nodes. This level is intended for intensive workloads with higher query performance and compute scalability. You can choose the pay-as-you-go model or save 37% to 65% by using reserved instances if you can commit to one or three years. For more information, see Reserved instances.

> Start with smaller DWUs and measure performance for resource intensive operations, such as heavy data loading or transformation. This will help you determine the number of units you need to increase or decrease. Measure usage during the peak business hours so you can assess the number of concurrent queries and accordingly add units to increase the parallelism. Conversely, measure off-peak usage so that you can pause compute when needed.

In Azure Synapse Analytics, you can import or export data from an external data store, such as Azure Blob Storage and Azure Data Lake Store. Storage and analytics resources aren't included in the price. There is additional bandwidth cost for moving data in and out of the data warehouse.

For more information, see these articles:

- Azure Synapse Pricing
- Manage compute in Azure Synapse Analytics data warehouse

Reference architecture

- Automated enterprise BI with Azure Synapse Analytics and Azure Data Factory
- Enterprise BI in Azure with Azure Synapse Analytics

Azure Databricks

Azure Databricks offers two SKUs **Standard** and **Premium**, each with these options, listed in the order of least to most expensive.

- **Data Engineering Light** is for data engineers to build and execute jobs in automated Spark clusters.
- **Data Engineering** includes autoscaling and has features for machine learning flows.
- **Data Analytics** includes the preceding set of features and is intended for data scientists to explore, visualize, manipulate, and share data and insights interactively.

Choose a SKU depending on your workload. If you need features like log audit, which is available in **Premium**, the overall cost can increase. If you need autoscaling of clusters to handle larger workloads or interactive Databricks dashboards, choose an option higher than **Data Engineering Light**.

Here are factors that impact Databricks billing:

- Azure location where the resource is provisioned.
- The virtual machine instance tier and number of hours the instances were running.
- *Databricks units (DBU)*, which is a unit of processing capability per hour, billed on per-second usage.

> The example is based on the current price and is subject to change. The calculation shown is for information purposes only.

Suppose you run a **Premium** cluster for 100 hours in East US 2 with 10 DS13v2 instances.

ITEM	EXAMPLE ESTIMATE
Cost for 10 DS13v2 instances	100 hours x 10 instances x $0.741/hour = $741.00
DBU cost for **Data Analytics** workload	100 hours x 10 instances x 2 DBU per node x $0.55/DBU = $1,100
Total	$1,841

For more information, see Azure Databricks Pricing.

If you can commit to one or three years, opt for reserved instances, which can save 38% - 59%. For more information, see Reserved instances.

> Turning off the Spark cluster when not in use to prevent unnecessary charges.

Reference architecture

- Stream processing with Azure Databricks
- Build a Real-time Recommendation API on Azure
- Batch scoring of Spark models on Azure Databricks

Azure Stream Analytics

Stream analytics uses *streaming units (SUs)* to measure the amount of compute, memory, and throughput required to process data. When provisioning a stream processing job, you're expected to specify an initial number of SUs. Higher streaming units mean higher cost because more resources are used.

Stream processing with low latency requires a significant amount of memory. This resource is tracked by the SU% utilization metric. Lower utilization indicates that the workload requires more compute resources. You can set an alert on 80% SU Utilization metric to prevent resource exhaustion.

> To evaluate the number of units you need, process an amount of data that is realistic for your production level workload, observe the SU% Utilization metric, and accordingly adjust the SU value.

You can create stream processing jobs in Azure Stream Analytics and deploy them to devices running Azure IoT Edge through Azure IoT Hub. The number of devices impacts over all cost. Billing starts when a job is deployed to devices, regardless of the job status (running, failed, stopped).

SUs are based on the partition configuration for the inputs and the query that's defined within the job. For more information, see Calculate the max streaming units for a job and Understand and adjust Streaming Units.

For pricing details, see Azure Stream Analytics pricing.

Reference architecture

- Batch scoring of Python machine learning models on Azure
- Azure IoT reference architecture

Azure Analysis Services

Big data solutions need to store data that can be used for reporting. Azure Analysis Services supports the creation of tabular models to meet this need.

Here are the tiers:

- **Developer** is recommended for evaluation, development, and test scenarios.
- **Basic** is recommended for small production environment.
- **Standard** is recommended for mission critical workloads.

Analysis Services uses Query Processing Units (QPUs) to determine the processing power. A QPU is an abstracted measure of compute and data processing resources that impact performance. Higher the QPU results in higher performance.

Each tier offers one or more instances. The main cost drivers are the QPUs and memory allocated for the tier instance. Start with a smaller instance, monitor the QPU usage, and scale up or down by selecting a higher or lower instance within the tier. Also, monitor usage during off-peak hours. You can pause the server when not in use. No charges apply when you pause your instance. For more information, see these articles:

- The right tier when you need it
- Monitor server metrics
- Azure Analysis Services pricing

Reference architecture

- Enterprise business intelligence - Azure Reference Architectures
- Automated enterprise BI - Azure Architecture Center

Azure Data Factory V2

Azure Data Factory is a big data orchestrator. The service transfers data to and from diverse types of data stores. It transforms the data by using other compute services. It creates workflows that automate data movement and transformation. You are only charged for consumption. Consumption is measured by these factors:

- Pipeline activities that take the actions on the data. Those actions include copying the data from various sources, transforming it, and controlling the flow. For more information, see data movement activities, data transformation activities, and control activities.

  You are charged for the total activities in thousands.

- Executions measured in data integration units. Each unit tracks CPU, memory, and network resource allocation. This measurement applies to Azure Integration Runtime.

You are also charged for execution activities, such as copying data, lookups, and external activities. Each activity is individually priced. You are also charged for pipelines with no associated triggers or runs within the month. All activities are prorated by the minute and rounded up.

Reference architecture

- Automated enterprise BI - Azure Architecture Center
- Enterprise business intelligence - Azure Reference Architectures
- Build an enterprise-grade conversational bot - Azure Architecture Center

Compute cost estimates

11/19/2021 • 8 minutes to read • Edit Online

Compute refers to the hosting model for the computing resources that your application runs on. Whether you're hosting model is Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Function as a service (FaaS), each resource requires your evaluation to understand the tradeoffs that can be made that impact your cost. To learn more about hosting models, read Understand the hosting models.

- **Infrastructure-as-a-Service** (IaaS) lets you provision individual virtual machines (VMs) along with the associated networking and storage components. Then you deploy whatever software and applications you want onto those VMs. This model is the closest to a traditional on-premises environment, except that Microsoft manages the infrastructure. You still manage the individual VMs.
- **Platform-as-a-Service** (PaaS) provides a managed hosting environment, where you can deploy your application without needing to manage VMs or networking resources. Azure App Service is a PaaS service.
- **Functions-as-a-Service** (FaaS) goes even further in removing the need to worry about the hosting environment. In a FaaS model, you simply deploy your code and the service automatically runs it. Azure Functions are a FaaS service.

What are the cost implications to consider for choosing a hosting model?

If your application can be broken down into short pieces of code, a FaaS hosting model might be the best choice. You're charged only for the time it takes to execute your code. For example, Azure Functions is a FaaS service that processes events with serverless code. Azure Functions allows you to run short pieces of code (called functions) without worrying about application infrastructure. Use one of the three Azure Functions pricing plans to fit your need. To learn more about the pricing plans, see How much does Functions cost?

If you want to deploy a larger or more complex application, PaaS may be the better choice. With PaaS, your application is always running, as opposed to FaaS, where your code is executed only when needed. Since more resources are used with PaaS, the price increases.

If you are migrating your infrastructure from on-premises to Azure, IaaS will greatly reduce and optimize infrastructure costs and salaries for IT staff who are no longer needed to manage the infrastructure. Since IaaS uses more resources than PaaS and FaaS, your cost could be highest.

What are the main cost drivers for Azure services?

You will be charged differently for each service depending on your region, licensing plan (e.g., Azure Hybrid Benefit for Windows Server), number and type of instances you need, operating system, lifespan, and other parameters required by the service. Assess the need for each compute service by using the flowchart in Choose a candidate service. Consider the tradeoffs that will impact your cost by creating different estimates using the Pricing Calculator. If your application consists of multiple workloads, we recommend that you evaluate each workload separately. See Consider limits and costs to perform a more detailed evaluation on service limits, cost, SLAs, and regional availability.

Are there payment options for Virtual Machines (VMs) to help meet my budget?

The best choice is driven by the business requirements of your workload. If you have higher SLA requirements, it will increase overall costs. You will likely need more VMs to ensure uptime and connectivity. Other factors that will impact cost are region, operating system, and the number of instances you choose. Your cost also depends

on the workload life span. See Virtual machines and Use Spot VMs in Azure for more details.

- **Pay as you go** lets you pay for compute capacity by the second, with no long-term commitment or upfront payments. This option allows you to increase or decrease compute capacity on demand. It is appropriate for applications with short-term, spiky, or unpredictable workloads that cannot be interrupted. You can also start or stop usage at any time, resulting in paying only for what you use.
- **Reserved Virtual Machine Instances** lets you purchase a VM for one or three years in a specified region in advance. It is appropriate for applications with steady-state usage. You may save more money compared to pay-as-you-go pricing.
- **Spot pricing** lets you purchase unused compute capacity at major discounts. If your workload can tolerate interruptions, and its execution time is flexible, then using spot pricing for VMs can significantly reduce the cost of running your workload in Azure.
- **Dev-Test pricing** offers discounted rates on Azure to support your ongoing development and testing. Dev-Test allows you to quickly create consistent development and test environments through a scalable, on-demand infrastructure. This will allow you to spin up what you need, when you need it, and explore scenarios before going into production. To learn more about Azure Dev-Test reduced rates, see Azure Dev/Test Pricing.

For details on available sizes and options for the Azure VMs you can use to run your apps and workloads, see Sizes for virtual machines in Azure. For details on specific Azure VM types, see Virtual Machine series.

Do I pay extra to run large-scale parallel and high-performance computing (HPC) batch jobs?

Use Azure Batch to run large-scale parallel and HPC batch jobs in Azure. You can save on VM cost because multiple apps can run on one VM. Configure your workload with either the Low-priority tier (the cheapest option) or the Standard tier (provides better CPU performance). There is no cost for the Azure Batch service. Charges accrue for the underlying resources that run your batch workloads.

Use PaaS as an alternative to buying VMs

When you use the IaaS model, you do have final control over the VMs. It may appear to be a cheaper option at first, but when you add operational and maintenance costs, the cost increases. When you use the PaaS model, these extra costs are included in the pricing. In some cases, this means that PaaS services can be a cheaper than managing VMs on your own. For help finding areas in the architecture where it may be natural to incorporate PaaS options, see Managed services.

How can I cut costs with hosting my web apps in PaaS?

If you host you web apps in PaaS, you'll need to choose an App Service plan to run your apps. The plan will define the set of compute resources on which your app will run. If you have more than one app, they will run using the same resources. This is where you will see the most significant cost savings, as you don't incur cost for VMs.

If your apps are event-driven with a short-lived process using a microservices architecture style, we recommend using Azure Functions. Your cost is determined by execution time and memory for a single function execution. For pricing details, see Azure Functions pricing.

Is it more cost-effective to deploy development testing (dev-test) on a PaaS or IaaS hosting model?

If your dev-test is built on Azure managed services such as Azure DevOps, Azure SQL Database, Azure Cache for Redis, and Application Insights, the cheapest solution might be using the PaaS hosting model. You won't incur the cost and maintenance of hardware. If your dev-test is built on Azure managed services such as Azure

DevOps, Azure DevTest Labs, VMs, and Application Insights, you need to add the cost of the VMs, which can greatly increase your cost. For details on evaluating, see Azure Dev/Test Pricing.

A special case of PaaS - Containers

How can I get the best cost savings for a containerized workload that requires full orchestration?

Your business requirements may necessitate that you store container images so that you have fast, scalable retrieval, and network-close deployment of container workloads. Although there are choices as to how you will run them, we recommend that you use AKS to set up instances with a minimum of three (3) nodes. AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure. There is no charge for AKS Cluster Management. Any additional costs are minimal. The containers themselves have no impact on cost. You pay only for per-second billing and custom machine sizes.

Can I save money if my containerized workload does not need full orchestration?

Your business requirements may not necessitate full orchestration. If this is the case and you are using App Service containers, we recommend that you use one of the App Service plans. Choose the appropriate plan based on your environment and workload.

There is no charge to use SNI-based SSL. Standard and Premium service plans include the right to use one IP SSL at no additional charge. Free and shared service plans do not support SSL. You can purchase the right to use additional SSL connections for a fee. In all cases the SSL certificate itself must be purchased separately. To learn more about each plan, see App services plans.

Where's the savings if my workload is event driven with a short-lived process?

In this example, Service Fabric may be a better choice than AKS. The biggest difference between the two is that AKS works only with Docker applications using Kubernetes. Service Fabric works with microservices and supports many different runtime strategies. Service Fabric can deploy Docker and Windows Server containers. Like AKS, Service Fabric is built for the microservice and event-driven architectures. AKS is strictly a service orchestrator and handles deployments, whereas Service Fabric also offers a development framework that allows building stateful/stateless applications.

Predict cost estimates using the Pricing Calculator

Use the Pricing Calculator to create your total cost estimate. After you run your initial scenario, you may find that your plan is beyond the scope of your budget. You can adjust the overall cost and create various cost scenarios to make sure your needs are met before you commit to purchasing.

> **NOTE**
> The costs in this example are based on the current price and are subject to change. The calculation is for information purposes only. It shows the Collapsed view of the cost in this estimate.

> **TIP**
> You can start building your cost estimate at any time and re-visit it later. The changes will be saved until you modify or delete your estimate.

Next steps

- Provisioning cloud resources to optimize cost
- Virtual Machines documentation
- VM payment options
- Pricing Calculator

Data store cost estimates

11/19/2021 • 9 minutes to read • Edit Online

Most cloud workloads adopt the *polyglot* persistence approach. Instead of using one data store service, a mix of technologies is used. You can achieve optimal cost benefit from using this approach.

Each Azure data store has a different billing model. To establish a total cost estimate, first, identify the business transactions and their requirements. Then, break each transaction into operations. Lastly, identify a data store appropriate for the type of data. Do this for each workload separately.

Let's take an example of an e-commerce application. It needs to store data for transactions such as orders, payments, and billing. The data structure is predetermined and not expected to change frequently. Data integrity and consistency are crucial. There's a need to store product catalogs, social media posts, and product reviews. In some cases, the data is unstructured, and is likely to change over time. Media files must be stored and data must be stored for auditing purposes.

Learn about data stores in Understand data store models.

Guidelines: Identify the business transactions and their requirements

The following list of questions address the requirements that can have the greatest impact your cost estimate. For example, your monthly bill may be within your budget now, but if you scale up or add storage space later, your cost may increase well over your budget.

- Will your data need to be migrated to on-premises, external data centers, or other cloud hosting environments?
- What type of data are you intending to store?
- How large are the entities you need to store?
- What is the overall amount of storage capacity you need?
- What kind of schemas will you apply to your data (e.g., fixed schema, schema-on-write, or schema-on-read)?
- What are your data performance requirements (e.g., acceptable response times for querying and aggregation of data once ingested)?
- What level of fault-tolerance do you need to provide for data consumers?
- What kind of data replication capabilities do you require?
- Will the limits of a particular data store support your requirements for scale, number of connections, and throughput?
- How many instances will need to run to support your uptime and throughput requirements? (Consider operations costs in this calculation.)
- Can you partition your data to store it more cost effectively (e.g., can you move large objects out of an expensive relational database into an object store)?

There are other requirements that may not have as great of an impact on your cost. For example, the US East region is only slightly lower than Canada Central. See Criteria for choosing a data store for additional business requirements.

Use the Pricing Calculator to determine different cost scenarios.

What are the network requirements that may impact cost?

- Do you need to restrict or otherwise manage access to your data from other network resources?

- Does data need to be accessible only from inside the Azure environment?
- Does the data need to be accessible from specific IP addresses or subnets?
- Does the data need to be accessible from applications or services hosted on-premises or in other external data centers?

Guidelines: Break each transaction into operations

For example, one business transaction can be processed by these distinct operations:

- 10-15 database calls
- three append operations on blob, and
- two list operations on two separate file shares

How does data type in each operation effect cost?

- Consider Azure Blob Storage Block Blobs instead of storing binary image data in Azure SQL Database. Blob storage is cheaper than Azure SQL Database.
- If your design requires SQL, store a lookup table in SQL Database and retrieve the document when needed to serve it to the user in your application middle tier. SQL Database is highly targeted for high-speed data lookups and set-based operations.
- The hot access tier of Azure Block Blob Storage cost is cheaper than the equivalent size of the Premium SSD volume that has the database.

Read this decision chart to make your choices.

Guidelines: Identify a data store appropriate for the type of data

> **NOTE**
> An inappropriate data store or one that is mis-configured can have a huge cost impact on your design.

Relational database management systems (RDBMS) cost

RDBMS is the recommended choice when you need strong consistency guarantees. An RDBMS typically supports a schema-on-write model, where the data structure is defined ahead of time, and all read or write operations must use the schema.

How can I save money if my data is on-premises and already on SQL server?

If the on-premises data is already on a SQL server, it might be a natural choice. The on-premises license with Software Assurance can be used to bring down the cost if the workload is eligible for Azure Hybrid Benefit. This option applies for Azure SQL Database (PaaS) and SQL Server on Azure Virtual Machines (IaaS).

For open-source databases such as MySQL, MariaDB, or PostGreSQL, Azure provides managed services that are easy to provision.

What are some design considerations that will affect cost?

- If the SLAs don't allow for downtime, can a read-only replica in a different region enable business continuity?
- If the database in the other region must be read/write, how will the data be replicated?
- Does the data need to be synchronous, or could consistency allow for asynchronous replication?
- How important is it for updates made in one node to appear in other nodes, before further changes can be made?

Azure storage has several options to make sure data is copied and available when needed. Locally redundant storage (LRS) synchronously replicates data in the primary region. If the entire primary center is unavailable the replicated data is lost. At the expensive end, Geo-zone-redundant storage (preview) (GZRS) replicates data in availability zones within the primary region and asynchronously copied to another paired region. Databases that offer geo-redundant storage, such as SQL Database, are more expensive. Most other OSS RDBMS database use LRS storage, which contributes to the lower price range.

For more information, see automated backups.

Cosmos DB offers five consistency levels: *strong*, *bounded staleness*, *session*, *consistent prefix*, and *eventual* consistency. Each level provides availability and performance tradeoffs and is backed by comprehensive SLAs. The consistency level itself does not affect cost.

How can I minimize compute cost?

Higher throughput and IOPS require higher compute, memory, I/O and storage limits. These limits are expressed in a vCore model. With higher vCore number, you buy more resources and consequently the cost is higher. Azure SQL Database has more vCores and allows you to scale in smaller increments. Azure Database for MySQL, PostgreSQL, and MariaDB have fewer vCores and scaling up to a higher vCore can cost more. MySQL provides in-memory tables in the **Memory Optimized** tier, which can also increase the cost.

All options offer a consumption and provisioned pricing models. With pre-provisioned instances, you save more if you can commit to one or three years.

How is primary and backup storage cost calculated?

With Azure SQL Database the initial 32 GB of storage is included in the price. For the other listed options, you need to buy storage separately and might increase the cost depending on your storage needs.

For most databases, there is no charge for the price of backup storage that is equal in size to primary storage. If you need more backup storage, you will incur an additional cost.

Key/value and document databases cost

Azure Cosmos DB is recommended for key/value stores and document databases. Azure Cache for Redis is also recommended for key/value stores.

For Cosmos DB, here are some considerations that will affect cost:

- Can storage and item size be adjusted?
- Can index policies be reduced or customized to bring down the extra cost for writes and eliminate the requirement for additional throughput capacity?
- If data is no longer needed, can it be deleted from the Azure Cosmos account? As an alternative, you can migrate the old data to another data store such as Azure blob storage or Azure data warehouse.

For Azure Cache for Redis, there is no upfront cost, no termination fees, you pay only for what you use, and billing is per-hour.

Graphic databases cost

Cost considerations for graphic database stores include storage required by data and indexes used across all the regions. For example, graphs need to scale beyond the capacity of a single server, and make it easy to lookup the index when processing a query.

The Azure service that supports this is Gremlin API in Azure Cosmos DB. Cost is limited to the Azure Cosmos DB usage. You pay for the operations you perform against the database and for the storage consumed by your data. Charges are determined by the number of provisioned containers, the number of hours the containers were

online, and the provisioned throughput for each container.

If the on-premises data is already on an SQL server on Azure Virtual Machines (IaaS), the license with Software Assurance can be used to bring down the cost if the workload is eligible for Azure Hybrid Benefit.

Data analytics cost

Cost considerations for data analytics stores include data storage, multiple servers to maximize scalability, and the ability to access large data as external tables.

Azure has many services that support data analytics stores: Azure Synapse Analytics, Azure Data Lake, Azure Data Explorer, Azure Analysis Services, HDInsight, and Azure Databricks. As an example of use, historical data is typically stored in data stores such as blob storage or Azure Data Lake Storage Gen2, which are then accessed by Azure Synapse, Databricks, or HDInsights as external tables.

When using Azure Synapse, you will only pay for the capabilities you opt in to use. During public preview, there will not be a cost for provisioning an Azure Synapse workspace. Enabling a managed VNET and Customer Managed Keys may incur a workspace fee after public preview. Pricing of workspaces with additional capabilities will be announced at a future date.

Column family database cost

The main cost consideration for a column family database is that it needs to be massively scalable.

The Azure services that support column family databases are Azure Cosmos DB Cassandra API and HBase in HDInsight.

For Azure Cosmos DB Cassandra API, you pay the cost of Cosmos DB, which includes database operations and consumed storage. Cassandra API is open-source.

For HBase in HDInsight, you pay the cost of HDInsight, which includes instance size and number. HBase is open-source.

Search engine database cost

Cost incurs for a search engine database when applications need to search for information held in external data stores. It also needs to index massive volumes of data and provide near real-time access to these indexes.

Azure Cognitive Search is a search service that uses AI capabilities to identify and explore relevant content at scale.

Time series database cost

The main cost considerations for a time series database is the need to collect large amounts of data in real time from a large number of sources. Although the records written to a time series database are generally small, there are often a large number of records, and total data size can grow rapidly which will drive up cost.

Azure Time Series Insights may be the best option to minimize cost.

Object storage cost

Cost considerations include storing large binary objects such as images, text files, video and audio streams, large application data objects and documents, and virtual machine disk images.

Azure services that support object storage are Azure Blob Storage and Azure Data Lake Storage Gen2.

Shared files cost

The main cost consideration is having the ability to access files across a network. For example, given appropriate security and concurrent access control mechanisms, sharing data in this way can enable distributed services to provide highly scalable data access for performing basic, low-level operations such as simple read and write requests.

The Azure service that supports shared files is Azure Files.

Azure messaging cost estimates

11/19/2021 • 2 minutes to read • Edit Online

The messaging services in this article have no up-front cost or termination fees, and you pay only for what you use. In some cases, it's advantageous to combine two messaging services to increase the efficiency of your messaging system. See Crossover scenarios for examples.

Cost is based on the number of operations or throughput units used depending on the message service. Using the wrong messaging service for the intent can lead to higher costs. Before choosing a service, first, determine the intent and requirements of the messages. Then, consider the tradeoffs between cost and operations/throughput units. For tradeoff examples, see Technology choices for a message broker.

Use the Azure Pricing calculator for help creating various cost scenarios.

Service Bus cost

Connect on-premises and cloud-based applications and services to implement highly secure messaging workflows using Service Bus. Cost is based on messaging operations and number of connections. The Basic tier is the cheapest. If you want more operations and features, choose the Standard or Premium tier. For example, Service Bus Premium runs in dedicated resources to provide higher throughput and more consistent performance.

For pricing details, see Service Bus pricing.

Event Grid cost

Manage routing of all events from any source to any destination to simplify event-based app development using Event Grid. Event Grid can route a massive number of events per second per region. Cost is based on number of operations performed. Examples of some operations are event ingress, subscription delivery attempts, management calls, and filtering by subject suffix.

For pricing details, see Event Grid pricing.

Event Hubs

Stream millions of events per second from any source to build dynamic data pipelines and immediately respond to business challenges using Event Hubs. Cost is based on throughput units. A key difference between Event Grid and Event Hubs is in the way event data is made available to subscribers. For more information, see Pull model.

For questions and answers on pricing, see Pricing.

For pricing, see Event Hubs pricing.

Networking cost estimates

11/19/2021 • 5 minutes to read • Edit Online

Traffic routing and load balancing

Most workloads have a load balancing service to route and distribute traffic in a way that a single resource isn't overloaded, and the response time is minimum with maximum throughput.

Do you need to distribute traffic within a region or across regions?

Azure Front Door and Traffic Manager can distribute traffic to backends, clouds, or hybrid on-premises services that reside in multiple regions. Otherwise, for regional traffic that moves within virtual networks or zonal and zone-redundant service endpoints within a region, choose Application Gateway or Azure Load Balancer.

What is the type of traffic?

Load balancers such as Azure Application Gateway and Azure Front Door are designed to route and distribute traffic to web applications or HTTP(S) endpoints. Those services support Layer 7 features such as SSL offload, web application firewall, path-based load balancing, and session affinity. For non-web traffic, choose Traffic Manager or Azure Load Balancer. Traffic Manager uses DNS to route traffic. Load Balancer supports Layer 4 features for all UDP and TCP protocols.

Here's the matrix for choosing load balancers by considering both dimensions.

SERVICE	GLOBAL/REGIONAL	RECOMMENDED TRAFFIC
Azure Front Door	Global	HTTP(S)
Traffic Manager	Global	non-HTTP(S)
Application Gateway	Regional	HTTP(S)
Azure Load Balancer	Regional	non-HTTP(S)

For more information, see Choose a load balancing service.

Example cost analysis

Consider a web application that receives traffic from users across regions over the internet. To minimize the request response time, the load balancer can delegate the responsibility from the web server to a different machine. The application is expected to consume 10 TB of data. Routing rules are required to route incoming requests to various paths. Also, we want to use Web Application Firewall (WAF) to secure the application through policies.

By using the Azure Pricing Calculator, we can estimate the cost for these two services.

> The example is based on the current price and is subject to change. The calculation shown is for information purposes only.

Azure Front Door (West US), Zone 1.

Azure Front Door

REGION:

West US

Outbound Data Transfer

Zone 1: North America, Europe and Africa

10 TB = $1,736.00

Inbound Data Transfer

10 TB = $102.40

Routing rules

20 Rules × 730 Hours = $240.90

Web Application Firewall (WAF)

Policy

50 Policies × $5.00 Per policy per month = $250.00

Custom Rules

20 Rules × $1.00 Per rule per month = $20.00

20 Requests processed (in millions) × $0.60 Per million requests = $12.00

Managed Ruleset

5 Default Rulesets × $20.00 Per ruleset per month = $100.00

10 Requests processed (in millions) × $1.00 Per million requests = $10.00

Upfront cost	$0.00
Monthly cost	$2,471.30

Application Gateway (West US)

Application Gateway

REGION: West US

TIER: Standard V2

Fixed Gateway Hours

730 Hours = $189.80

Capacity unit

10 Compute unit(s)

1500 Persistent Connection(s)

2 Throughput (mb/s)

Each capacity unit is composed of at most: 1 compute unit, or 2,500 persistent connections, or 2.22-Mbps throughput. If any one of these metrics are exceeded, then another n capacity unit(s) are necessary, even if the other two metrics don't exceed this single capacity unit's limits.

730 Hours = $58.40

No corresponding zone for this region

10 TB = $890.44

Upfront cost $0.00

Monthly cost $1,138.65

Consider a similar example where the type of traffic is changed. Instead the application is a UDP streaming service that is deployed across regions and traffic goes over the internet. We can use a combination of Traffic Manager and Azure Load Balancer. Traffic Manager is a simple routing service that uses DNS to direct clients to specific service. Here are the cost estimates:

Azure Traffic Manager

ITEM	EXAMPLE ESTIMATE
DNS queries	100 million x $0.54/per million = $54.00
Basic health checks	20 endpoints x $0.36/month = $7.20
Fast interval health checks	10 endpoints x $1.00/month = $10.00
External endpoints hosted outside Azure	10 endpoints x $0.54/month = $5.40
Real User Measurements	100 million user measurements x $2.00/month = $200.00
Traffic view	100 million data points processed x $2.00/month = $200.00
Total	$486.60

Azure Load Balancer

ITEM	EXAMPLE ESTIMATE
Rules	First 5 rules: $0.032/rule/hour x 730 = $18.25
	Additional 5 rules: $0.013/rule/hour x 730 = $36.50
Data processed	10 TB x $0.005/GB = 51.20

ITEM	EXAMPLE ESTIMATE
Total	$105.95

Peering

Do you need to connect virtual networks?

Peering technology is a service used for Azure virtual networks to connect with other virtual networks in the same or different Azure region. Peering technology is used often in hub and spoke architectures.

An important consideration is the additional costs incurred by peering connections on both egress and ingress traffic traversing the peering connections.

> Keeping the top talking services of a workload within the same virtual network, zone and/or region unless otherwise required. Use virtual networks as shared resources for multiple workloads against a single virtual network per workload approach. This approach will localize traffic to a single virtual network and avoid the additional costs on peering charges.

Example cost analysis

> The example is based on the current price and is subject to change. The calculation shown is for information purposes only.

Peering within the same region is cheaper than peering between regions or Global regions. For instance, you consume 50 TB per month by connecting two VNETs in Central US. Using the current price here is the incurred cost.

ITEM	EXAMPLE ESTIMATE
Ingress traffic	50 TB x $0.0100/GB = $512.50
Egress traffic	50 TB x $0.0100/GB = $512.50
Total	$1,025.00

Let's compare that cost for cross-region peering between Central US and East US.

ITEM	EXAMPLE ESTIMATE
Ingress traffic	50 TB x $0.0350/GB = $1,792.00
Egress traffic	50 TB x $0.0350/GB = $1,792.00
Total	$3,584.00

Hybrid connectivity

There are two main options for connecting an on-premises datacenter to Azure datacenters:

- Azure VPN Gateway can be used to connect a virtual network to an on-premises network through a VPN appliance or to Azure Stack through a site-to-site VPN tunnel.

- Azure ExpressRoute creates private connections between Azure datacenters and infrastructure that's on premise or in a colocation environment.

What is the required throughput for cross-premises connectivity?

VPN gateway is recommended for development/test cases or small-scale production workloads where throughput is less than 100 Mbps. Use ExpressRoute for enterprise and mission-critical workloads that access most Azure services. You can choose bandwidth from 50 Mbps to 10 Gbps.

Another consideration is security. Unlike VPN Gateway traffic, ExpressRoute connections don't go over the public internet. VPN Gateway traffic is secured by industry standard IPsec.

For both services, inbound transfers are free and outbound transfers are billed per the billing zone.

For more information, see Choose a solution for connecting an on-premises network to Azure.

This blog post provides a comparison of the two services.

Example cost analysis

This example compares the pricing details for VPN Gateway and ExpressRoute.

> The example is based on the current price and is subject to change. The calculation shown is for information purposes only.

Azure VPN Gateway

Suppose, you choose the **VpnGw2AZ** tier, which supports availability zones; 1 GB/s bandwidth. The workload needs 15 site-to-site tunnels and 1 TB of outbound transfer. The gateway is provisioned and available for 720 hours.

ITEM	EXAMPLE ESTIMATE
VPN Hours	720 x $0.564 = $406.08
Site to Site (S2S) Tunnels	The first 10 tunnels are free. For additional tunnels, the cost is 5 tunnels x 720 hours x $0.015 per hour per tunnel = $54.00
Outbound transfer	1024 GB x 0.086 = $88.65
Total cost per month	$548.73

ExpressRoute

Let's choose the **Metered Data** plan in billing zone of **Zone 1**.

ITEM	EXAMPLE ESTIMATE
Circuit bandwidth	1 GB/s speed has a fixed rate of $436.00 for the standard price.
Outbound transfer	1 TB x $0.0025 = $25.60
Total cost per month	$461.00

The main cost driver is outbound data transfer. ExpressRoute is more cost-effective than VPN Gateway when consuming large amounts of outbound data. If you consume more than 200 TB per month, consider ExpressRoute with the **Unlimited Data** plan where you're charged a flat rate.

For pricing details, see:

- Azure VPN Gateway pricing
- Azure ExpressRoute pricing

Networking resources provisioning

11/19/2021 • 7 minutes to read • Edit Online

For design considerations, see Networking resource choices.

Azure Front Door

Azure Front Door billing is affected by outbound data transfers, inbound data transfers, and routing rules. The pricing chart doesn't include the cost of accessing data from the backend services and transferring to Front Door. Those costs are billed based on data transfer charges, described in Bandwidth Pricing Details.

Another consideration is Web Application Firewall (WAF) settings. Adding policies will drive up the cost.

For more information, see Azure Front Door Pricing.

Reference architecture

Highly available multi-region web application uses Front Door to route incoming requests to the primary region. If the application running that region becomes unavailable, Front Door fails over to the secondary region.

Azure Application Gateway

There are two main pricing models:

- Fixed price

  You're charged for the time that the gateway is provisioned and available and the amount of data processed by the gateway. For more information, see Application Gateway pricing.

- Consumption price

  This model applies to v2 SKUs that offer additional features such as autoscaling, Azure Kubernetes Service Ingress Controller, zone redundancy, and others. You're charged based on the consumed capacity units. The capacity units measure the compute resources, persistent connections, and throughput. Consumption price is charged in addition to the fixed price.

For more information, see:

- Application Gateway v2 pricing
- Application Gateway pricing

Reference architecture

- Microservices architecture on Azure Kubernetes Service (AKS) uses Application Gateway as the ingress controller.

- Securely managed web applications uses Application Gateway as a web traffic load balancer operating at Layer 7 that manages traffic to the web application. Web Application Firewall (WAF) is enabled to enhance security.

Azure ExpressRoute

There are two main pricing models:

- **Metered Data plan**

There are two pricing tiers: **Standard** and **Premium**, which is priced higher. The tier pricing is based on the circuit bandwidth.

If you don't need to access the services globally, choose **Standard**. With this tier, you can connect to regions within the same zone at no additional cost. Outbound cross-zonal traffic incurs more cost.

- **Unlimited Data plan**

  All inbound and outbound data transfer is included in the flat rate. There are two pricing tiers: **Standard** and **Premium**, which is priced higher.

Calculate your utilization and choose a billing plan. The **Unlimited Data plan** is recommended if you exceed about 68% of utilization.

For more information, see Azure ExpressRoute pricing.

Reference architecture

Connect an on-premises network to Azure using ExpressRoute connects an Azure virtual network and an on-premises network connected using with VPN gateway failover.

Azure Firewall

Azure Firewall usage can be charged at a fixed rate per deployment hour. There's additional cost for the amount of data transferred.

There aren't additional cost for a firewall deployed in an availability zone. There are additional costs for inbound and outbound data transfers associated with availability zones.

When compared to network virtual appliances (NVAs), with Azure Firewall you can save up to 30-50%. For more information see Azure Firewall vs NVA.

Reference architecture

- Hub-spoke network topology in Azure
- Deploy highly available NVAs

Azure Load Balancer

The service distributes inbound traffic according to the configured rules.

There are two tiers: **Basic** and **Standard**.

The **Basic** tier is free.

For the **Standard** tier, you are charged only for the number of configured load-balancing and outbound rules. Inbound NAT rules are free. There's no hourly charge for the load balancer when no rules are configured.

See Azure Load Balancer Pricing for more information.

Reference architecture

- Connect an on-premises network to Azure using ExpressRoute: Multiple subnets are connected through Azure load balancers.
- SAP S/4HANA in Linux on Azure: Distribute traffic to virtual machines in the application-tier subnet.
- Extend an on-premises network using VPN Internal load balancer. Network traffic from the VPN gateway is routed to the cloud application through an internal load balancer. The load balancer is located in the front-end subnet of the application.

Azure VPN Gateway

When provisioning a VPN Gateway resource, choose between two gateway types:

- VPN gateway to send encrypted traffic across the public internet. Site-to-Site, Point-to-Site, and VNet-to-VNet connections all use a VPN gateway.
- ExpressRoute gateway - To send network traffic on a private connection. This is used when configuring Azure ExpressRoute.

For VPN gateway, select **Route-based** or **Policy-based** based on your VPN device and the kind of VPN connection you want to create. Route-based gateway allows point-to-site, inter-virtual network, or multiple site-to-site connections. Policy based only allows one site-to-site tunnel. Point-to-site isn't supported. So, route-based VPN is more expensive.

Next, you need to choose the SKU for **Route-based** VPN. For developer/test workloads, use **Basic**. For production workloads, an appropriate **Generation1** or **Generation2** SKU. Each SKU has a range and pricing depends on the type of VPN gateway because each type offers different levels of bandwidth, site-to-site, and point-to-site tunnel options. Some of those types also offer availability zones, which are more expensive. If you need higher bandwidth, consider Azure ExpressRoute.

VPN gateway can be the cost driver in a workload because charges are based on the amount of time that the gateway is provisioned and available.

All inbound traffic is free, all outbound traffic is charged as per the bandwidth of the VPN type. Bandwidth also varies depending on the billing zone.

For more information, see

- Hybrid connectivity
- VPN Gateway Pricing.
- Traffic across zones
- Bandwidth Pricing Details.

Reference architecture

- Extend an on-premises network using VPN connects the virtual network to the on-premises network through a VPN device.

Traffic Manager

Traffic manager uses DNS to route and load balance traffic to service endpoints in different Azure regions. So, an important use case is disaster recovery. In a workload, you can use Traffic Manager to route incoming requests to the primary region. If that region becomes unavailable, Traffic Manager fails over to the secondary region. There are other features that can make the application highly responsive and available. Those features cost money.

- Determine the best web app to handle request based on geographic location.
- Configure caching to reduce the response time.

Traffic Manager isn't charged for bandwidth consumption. Billing is based on the number of DNS queries received, with a discount for services receiving more than 1 billion monthly queries. You're also charged for each monitored endpoint.

Reference architecture

Multi-region N-tier application uses Traffic Manager to route incoming requests to the primary region. If that region becomes unavailable, Traffic Manager fails over to the secondary region. For more information, see the section Traffic Manager configuration.

DNS query charges

Traffic Manager uses DNS to direct clients to specific service.

Only DNS queries that reach Traffic Manager are charged in million query units. For 100 million DNS queries month, the charges will be $54.00 a month based on the current Traffic Manager pricing.

Not all DNS queries reach Traffic Manager. Recursive DNS servers run by enterprises and ISPs first attempt to resolve the query by using cached DNS responses. Those servers query Traffic Manager at a regular interval to get updated DNS entries. That interval value or TTL is configurable in seconds. TTL can impact cost. Longer TTL increases the amount of caching and reduces DNS query charges. Conversely, shorter TTL results in more queries.

However, there is a tradeoff. Increased caching also impacts how often the endpoint status is refreshed. For example, the user failover times, for an endpoint failure, will become longer.

Health monitoring charges

When Traffic Manager receives a DNS request, it chooses an available endpoint based on configured state and health of the endpoint. To do this, Traffic Manager continually monitors the health of each service endpoint.

The number of monitored endpoints are charged. You can add endpoints for services hosted in Azure and then add on endpoints for services hosted on-premises or with a different hosting provider. The external endpoints are more expensive, but health checks can provide high-availability applications that are resilient to endpoint failure, including Azure region failures.

Real User Measurement charges

Real User Measurements evaluates network latency from the client applications to Azure regions. That influences Traffic Manager to select the best Azure region in which the application is hosted. The number of measurements sent to Traffic Manager is billed.

Traffic view charges

By using Traffic View, you can get insight into the traffic patterns where you have endpoints. The charges are based on the number of data points used to create the insights presented.

Virtual Network

Azure Virtual Network is free. You can create up to 50 virtual networks across all regions within a subscription. Here are a few considerations:

- Inbound and outbound data transfers are charged per the billing zone. Traffic that moves across regions and billing zones are more expensive. For more information, see:
  - Traffic across zones
  - Bandwidth Pricing Details.
- VNET Peering has additional cost. Peering within the same region is cheaper than peering between regions or Global regions. Inbound and outbound traffic is charged at both ends of the peered networks. For more information, see Peering
- Managed services (PaaS) don't always need a virtual network. The cost of networking is included in the service cost.

Web application cost estimates

11/19/2021 • 4 minutes to read • Edit Online

All web applications (apps) have no up-front cost or termination fees. Some charge only for what you use and others charge per-second or per-hour. In addition, all web apps run in App Service plans. Together these costs can help determine the total cost of a web app.

Use the Azure Pricing calculator to help create various cost scenarios.

App Service plans

The App Service plans include the set of compute resources needed for the web app to run. Except for Free tier, an App Service plan carries a charge on the compute resources it uses. For a description of App Service plans, see Azure App Service plan overview.

You can potentially save money by choosing one or more App Service plans. To help find the solution that meets your business requirements, see Should I put an app in a new plan or an existing plan?

The *pricing tier* of an App Service plan determines what App Service features you get and how much you pay for the plan. The higher the tier, the higher the cost. For details on the pricing tiers, see App Service Pricing.

Are there exceptions that impact App Services plan cost?

You don't get charged for using the App Service features that are available to you (such as configuring custom domains, TLS/SSL certificates, deployment slots, backups, and so on). For a list of exceptions, see How much does my App Service plan cost?

App Service cost

App Service plans are billed on a per second basis. If your solution includes several App Service apps, consider deploying them to separate App Service plans. This approach enables you to scale them independently because they run on separate instances, which saves unnecessary cost.

Azure App Service supports two types of SSL connections: Server Name Indication (SNI) SSL Connections and IP Address SSL Connections. SNI-based SSL works on modern browsers while IP-based SSL works on all browsers. If your business requirements allow, use SNI-based SSL instead of IP-based SSL. There is no charge for SNI-based. IP-based SSL incurs a cost per connection.

See the SSL Connections section in App Service pricing for details.

API Management cost

If you want to publish APIs hosted on Azure, on-premises, and in other clouds more securely, reliably, and at scale, use API Management. The pricing tier you choose depends on the features needed based on the business requirements. For example, depending on your requirements for scalability, the number of units required to scale out range from 1-10. As the number of units increases, the cost of the pricing tier increases.

Self-hosted gateways are available in the Developer and Premium pricing tiers. There is an additional cost for this service using the Premium tier. There is no cost for self-hosted gateways if using the Developer tier.

For pricing details, see API Management Pricing.

Content Delivery Network (CDN) cost

If you want to offer users optimal online content delivery, use CDN in your workload. The data size being used has the most significant impact on cost, so you'll want to choose based on your business requirements. Purchasing data in increments of TB is significantly higher than purchasing data in increments of GB.

The provider you choose also can impact cost. Choose Microsoft as the provider to get the lowest cost.

Some data, such as shopping carts, search results, and other dynamic content, is not cacheable. CDN offers Acceleration Data Transfers, also called Dynamic Site Acceleration (DSA), to accelerate this type of web content. The price for DSA is the same across Standard and Premium profiles.

For pricing details, see Content Delivery Network pricing.

Azure Cognitive Search cost

To set up and scale a search experience quickly and cost-effectively, use Azure Cognitive Search. When you create an Azure Cognitive Search service, a resource is created at a pricing tier (or SKU) that's fixed for the lifetime of the service. Billing depends on the type of service. See Search API for a list of services.

The charges are based on the number of transactions for each type of operation specific to a service. A certain number of transactions are free. If you need additional transactions, choose from the Standard instances.

For pricing details, see Azure Cognitive Search pricing.

Azure SignalR cost

Use SignalR for any scenario that requires pushing data from server to client in real time. For example, Azure SignalR provides secure and simplified communication between client and app one-to-one (such as a chat window) or one-to-many (instant broadcasting, IoT dashboards, or notification to social network). To learn more about Azure SignalR, see What is Azure SignalR Service?

The Free tier is not recommended for a production environment. With the Standard tier, you pay only for what you use. A Standard tier is recommended as an enterprise solution because it offers a large number of concurrent connections and messages.

For pricing details, see Azure SignalR Service pricing.

Notification Hubs cost

To broadcast push notifications to millions of users at once, or tailor notifications to individual users, use Notification Hubs. Pricing is based on number of pushes. The Free tier is a good starting point for exploring push capabilities but is not recommended for production apps. If you require more pushes and features such as scheduled pushes or multi-tenancy, you can buy additional pushes per million for a fee.

See Push notifications with Azure Notification Hubs: Frequently asked questions for more information. For pricing details, see Notification Hubs pricing.

Checklist - Monitor cost

11/19/2021 • 2 minutes to read • Edit Online

Use this checklist to monitor the cost of the workload.

- **Gather cost data from diverse sources to create reports**. Start with tools like Azure Advisor, Advisor Score, and Azure Cost Management. Build custom reports relevant for the business by using Consumption APIs.
  - Cost reports
  - Review costs in cost analysis
- **Use resource tag policies to build reports**. Tags can be used to identify the owners of systems or applications and create custom reports.
  - Follow a consistent tagging standard
  - Video: How to review tag policies with Azure Cost Management
- **Use Azure built-in roles for cost**. Only give access to users who are intended to view and analyze cost reports. The roles are defined per scope. For example, use the **Cost Management Reader role** to enable users to view costs for their resources in subscriptions or resource groups.
  - Provide the right level of cost access
  - Azure RBAC scopes
- **Respond to alerts and have a response plan according to the constraints.** Respond to alerts quickly and identify possible causes and any required action.
  - Budget and alerts
  - Use cost alerts to monitor usage and spending
- **Adopt both proactive and reactive approaches for cost reviews**. Conduct cost reviews at a regular cadence to determine the cost trend. Also review reports that are created because of alerts.
  - Conduct cost reviews
  - Participate in central governance cost reviews
- **Analyze the cost at all scopes** by using Cost analysis. Identify services that are driving the cost through different dimensions, such as location, usage meters, and so on. Review whether certain optimizations are bringing results. For example, analyze costs associated with reserved instances and Spot VMs against business goals.
  - Quickstart: Explore and analyze costs with cost analysis
- **Detect anomalies** and identify changes in business or applications that might have contributed changes in cost. Focus on these factors:
  - Traffic pattern as the application scales.
  - Budget for the usage meters on resources.
  - Performance bottle necks.
  - CPU utilization and network throughput.
  - Storage footprint for blobs, backups, archiving.
- **Use Visualization tools to analyze cost information.**
  - Create visuals and reports with the Azure Cost Management connector in Power BI Desktop
  - Cost Management App

Set budgets and alerts

11/19/2021 • 2 minutes to read • Edit Online

Azure Cost Management has an alert feature. Alerts are generated when consumption reaches a threshold.

Consider the metrics that each resource in the workload. For each metric, build alerts on baseline thresholds. This way, the admins can be alerted when the workload is using the services at capacity. The admins can then tune the resources to target SKUs based on current load.

You can also set alerts on allowed budgets at the resource group or management groups scopes. Both cloud services performance and budget requirements can be balanced through alerts on metrics and budgets.

Over time, the workload can be optimized to autoheal itself when alerts are triggered. For information about using alerts, see Use cost alerts to monitor usage and spending.

Respond to alerts

When you receive an alert, check the current consumption data. Budget alerts aren't triggered in real time. There may be a delay between the alert and the current actual cost. Look for significant difference between cost values when the alert happened and the current cost. Next, conduct a cost review to discuss the cost trend, possible causes, and any required action. For information about stakeholders in a cost review, see Cost reviews.

Determine short and long-term actions justified by business value. Can a temporary increase in the alert threshold be a feasible fix? Does the budget need to be increased longer-term? Any increase in budget must be approved.

If the alert was caused because of unnecessary or expensive resources, you can implement additional Azure Policy controls. You can also add budget automation to trigger resource scaling or shutdowns.

Revise budgets

After you identify and analyze your spending patterns, you can set budget limits for applications or business units. You'll want to assign access to view or manage each budget to the appropriate groups. Setting several alert thresholds for each budget can help track your burn down rate.

Generate cost reports

11/19/2021 • 3 minutes to read • Edit Online

To monitor the cost of the workload, use Azure cost tools or custom reports. The reports can be scoped to business units, applications, IT infrastructure shared services, and so on. Make sure that the information is consistently shared with the stakeholders.

Azure cost tools

Azure provides cost tools that can help track cloud spend and make recommendations.

- Azure Advisor
- Advisor Score
- Azure Cost Management

Cost analysis

Cost analysis is a tool in Azure Cost Management that allows you to view aggregated costs over a period. This view can help you understand your spending trends.

View costs at different scopes, such as for a resource group or specific resource tags. Cost Analysis provides built-in charts and custom views. You can also download the cost data in CSV format to analyze with other tools.

For more information, see: Quickstart: Explore and analyze costs with cost analysis.

> **NOTE**
>
> There are many ways of purchasing Azure services. Not all are supported by Azure Cost Management. For example, detailed billing information for services purchased through a Cloud Solution Provider (CSP) must be obtained directly from the CSP. For more information about the supported cost data, see Understand cost management data.

Advisor recommendations

Azure Advisor recommendations for cost can highlight the over-provisioned services and ways to lower cost. For example, the virtual machines that should be resized to a lower SKU, unprovisioned ExpressRoute circuits, and idle virtual network gateways.

For more information, see Advisor cost management recommendations.

Consumption APIs

Granular and custom reports can help track cost over time. Azure provides a set of Consumption APIs to generate such reports. These APIs allow you to query and create various cost data. Data includes usage data for Azure services and third-party services through Marketplace, balances, budgets, recommendations on reserved instances, among others. You can configure Azure role-based access control (Azure RBAC) policies to allow only a certain set of users or applications access the data.

For example, you want to determine the cost of all resources used in your workload for a given period. One way of getting this data is by querying usage meters and the rate of those meters. You also need to know the billing period of the usage. By combining these APIs, you can estimate the consumption cost.

- Billing account API: To get your billing account to manage your invoices, payments, and track costs.
- Billing Periods API: To get billing periods that have consumption data.

- Usage Detail API: To get the breakdown of consumed quantities and estimated charges.
- Marketplace Store Charge API: To get usage-based marketplace charges for third-party services.
- Price Sheet API: To get the applicable rate for each meter.

The result of the APIs can be imported into analysis tools.

> **NOTE**
> Consumption APIs are supported for Enterprise Enrollments and Web Direct Subscriptions (with exceptions). Check Consumption APIs for updates to determine support for other types of Azure subscriptions.

For more information about common cost scenarios, see Billing automation scenarios.

Custom scripts

Use Azure APIs to schedule custom scripts that identify orphaned or empty resources. For example, unattached managed disks, load balancers, application gateways, or Azure SQL Servers with no databases. These resources incur a flat monthly charge while unused. Other resources may be stale, for example VM diagnostics data in blob or table storage. To determine if the item should be deleted, check its last use and modification timestamps.

Analyze and visualize

Start with the usage details in the invoice. Review that information against relevant business data and events. If there are anomalies, evaluate the significant changes in business or applications that might have contributed those changes.

Power BI Desktop has a connector that allows you to connect billing data from Azure Cost Management. You can create custom dashboards and reports, ask questions of your data and publish, and share your work.

> **NOTE**
> Sharing requires Power BI Premium licenses.

For more information, see Create visuals and reports with the Azure Cost Management connector in Power BI Desktop.

You can also use the Cost Management App. The app uses Azure Cost Management Template app for Power BI. You can import and analyze usage data and incurred cost within Power BI.

Conduct cost reviews

11/19/2021 • 2 minutes to read • Edit Online

The goal of cost monitoring is to review cloud spend with the intent of establishing cost controls and preventing any misuse. The organization should adopt both proactive and reactive review approaches for monitoring cost. Effective cost reviews must be conducted by accountable stakeholders at a regular cadence. The reviews must be complemented with reactive cost reviews, for example when a budget limit causes an alert.

Who should be included in a cost review?

The financial stakeholders must understand cloud billing to derive business benefits using financial metrics to make effective decisions.

Also include members of the technical team. Application owners, systems administrators who monitor and back-up cloud systems, and business unit representatives must be aware of the decisions. They can provide insights because they understand cloud cost metering and cloud architecture. One way of identifying owners of systems or applications is through resource tags.

What should be the cadence of cost reviews?

Cost reviews can be conducted as part of the regular business reviews. It's recommended that such reviews are scheduled,

- During the billing period. This review is to create an awareness of the estimated pending billing. These reports can be based on Azure Advisor, Advisor Score, and Azure Cost Management – Cost analysis.
- After the billing period. This review shows the actual cost with activity for that month. Use Balance APIs to generate monthly reports. The APIs can query data that gets information on balances, new purchases, Azure Marketplace service charges, adjustments, and overage charges.
- Because of a budget alert or Azure Advisor recommendations.

Web Direct (pay-as-you-go) and Cloud Solution Provider (CSP) billing occurs monthly. While Enterprise Agreement (EA) billing occurs annually, costs should still be reviewed monthly.

Checklist - Optimize cost

11/19/2021 • 2 minutes to read • Edit Online

Continue to monitor and optimize the workload by using the right resources and sizes. Use this checklist to optimize a workload.

- **Review the underutilized resources**. Evaluate CPU utilization and network throughput over time to check if the resources are used adequately. Azure Advisor identifies underutilized virtual machines. You can choose to decommission, resize, or shut down the machine to meet the cost requirements.
  - Resize virtual machines
  - Shutdown the under utilized instances
- **Continuously take action on the cost reviews**. Treat cost optimization as a process, rather than a point-in-time activity. Use tooling in Azure that provides recommendations on usage or cost optimization. Review the cost management recommendations and take action. Make sure that all stakeholders are in agreement about the implementation and timing of the change.
  - **Recommended** tab in the Azure portal
  - Recommendations in the Cost Management Power BI app
  - Recommendations in Azure Advisor
  - Recommendations using Reservation REST APIs
- **Use reserved instances on long running workloads**. Reserve a prepaid capacity for a period, generally one or three years. With reserved instances, there's a significant discount when compared with pay-as-you-go pricing.
  - Reserved instances
- **Use discount prices**. These methods of buying Azure resources can lower costs.
  - Azure Hybrid Use Benefit
  - Azure Reservations

  There are also payment plans offered at a lower cost:
  - Microsoft Azure Enterprise Agreement
  - Enterprise Dev Test Subscription
  - Cloud Service Provider (Partner Program)
- **Have a scale-in and scale-out policy**. In a cost-optimized architecture, costs scale linearly with demand. Increasing customer base shouldn't require more investment in infrastructure. Conversely, if demand drops, scale-down of unused resources. Autoscale Azure resources when possible.
  - Autoscale instances
- **Reevaluate design choices**. Analyze the cost reports and forecast the capacity needs. You might need to change some design choices.
  - **Choose the right storage tier**. Consider using hot, cold, archive tier for storage account data. Storage accounts can provide automated tiering and lifecycle management. For more information, see Review your storage options
  - **Choose the right data store**. Instead of using one data store service, use a mix of data store depending on the type of data you need to store for each workload. For more information, see Choose the right data store.

- **Choose Spot VMs for low priority workloads**. Spot VMs are ideal for workloads that can be interrupted, such as highly parallel batch processing jobs.
  - Spot VMs
- **Optimize data transfer**. Only deploy to multiple regions if your service levels require it for either availability or geo-distribution. Data going out of Azure datacenters can add cost because pricing is based on Billing Zones.
  - Traffic across billing zones and regions
- **Reduce load on servers**. Use Azure Content Delivery Network (CDN) and caching service to reduce load on front-end servers. Caching is suitable for servers that are continually rendering dynamic content that doesn't change frequently.
- **Use managed services**. Measure the cost of maintaining infrastructure and replace it with Azure PaaS or SaaS services.
  - Managed services

Autoscale instances

11/19/2021 • 2 minutes to read • Edit Online

In Azure, it's easier to grow a service with little to no downtime against downscaling a service, which usually requires deprovisioning or downtime. In general, opt for scale-out instead of scale up.

For certain application, capacity requirements may swing over time. Autoscaling policies allow for less error-prone operations and cost savings through robust automation.

Virtual machine instances

For auto scaling, consider the choice of instance size. The size can significantly change the cost of your workload.

> Choose smaller instances where workload is highly variable and scale out to get the desired level of performance, rather than up. This choice will enable you to make your cost calculations and estimates granular.

	1	25	50	75	100	125	150	175	200	225	250	275	300	325	350	375	400	425	450	475	500
4-core VM ~200 users	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3
Cost (Auto-scale 4 core VM)	584	584	584	584	584	584	584	584	584	1168	1168	1168	1168	1168	1168	1168	1168	1752	1752	1752	1752
1-core VM ~50 users	1	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10
Cost (Auto-scale 1 core VM)	146	146	146	292	292	438	438	584	584	730	730	876	876	1022	1022	1168	1168	1314	1314	1460	1460

Stateless applications

Many Azure services can be used to improve the application's ability to scale dynamically. Even where they may not have been originally designed to do so.

For example, many ASP.NET stateful web applications can be made stateless. Then they can be auto scaled, which results in cost benefit. You store state using Azure Redis Cache, or Cosmos DB as a back-end session state store through a Session State Provider.

Reserved instances

11/19/2021 • 2 minutes to read • Edit Online

Azure Reservations are offered on many services as a way to lower cost. You reserve a prepaid capacity for a period, generally one or three years. With reserved instances, there's a significant discount when compared with pay-as-you-go pricing. You can pay up front or monthly, price wise both are same.

Usage pattern

Before opting for reserved instances, analyze the usage data with pay-as-you-go prices over a time duration.

Do the services in the workload run intermittently or follow long-term patterns?

Azure provides several options that can help analyze usage and make recommendations by comparing reservations prices with pay-as-you-go price. An easy way is to view the **Recommended** tab in the Azure portal. Azure Advisor also provides recommendations that are applicable to an entire subscription. If you need a programmatic way, use the Reservation Recommendations REST APIs.

Reserved instances can lower cost for long running workloads. For intermittent workloads, prepaid capacity might go unused and it doesn't carry over to the next billing period. Usage exceeding the reserved quantity is charged using more expensive pay-as-you-go rates. But there are some flexible options. You can exchange or refund a reservation within limits. For more information, see Self-service exchanges and refunds for Azure Reservations.

Scope

Reservations can be applied to a specific scope—subscription, resource group, or a single resource. Suppose you set the scope as the resource group, the reservation savings will apply to the applicable resources provisioned in that group. For more information, see Scope reservations.

Services, subscription, and offer types

Many services are eligible for reservations. This range covers virtual machines and a wide variety of managed services. For information about the services that are eligible for reservations, see Charges covered by reservation.

Certain subscriptions and offer types support reservations. For a list of such subscriptions, see Discounted subscription and offer types.

Virtual machines

11/19/2021 • 3 minutes to read • Edit Online

Virtual machines can be deployed in fix-sized blocks. These VMs must be adequately sized to meet capacity demand and reduce waste.

For example, look at a VM running the SAP on Azure project can show you how initially the VM was sized based on the size of existing hardware server (with cost around €1 K per month), but the real utilization of VM was not more than 25% - but simple choosing the right VM size in the cloud we can achieve 75% saving (resize saving). And by applying the snoozing you can get additional 14% of economy:

It is easy to handle cost comparison when you are well equipped and for this Microsoft provides the set of specific services and tools that help you to understand and plan costs. These include the TCO Calculator, Azure Pricing Calculator, Azure Cost Management (Cloudyn), Azure Migrate, Cosmos DB Sizing Calculator, and the Azure Site Recovery Deployment Planner.

Here are some strategies that you can use to lower cost for virtual machines.

Resize virtual machines

You can lower cost by managing the size and the number of VMs.

> Determine the load by analyzing the CPU utilization to make sure that the instance is adequately utilized.

Ideally, with the right size, the current load should fit in a lower SKU of the same tier. Another way is to lower the number instances and still keep the load below a reasonable utilization. Azure Advisor recommends load less than 80% utilization for non-user facing workloads and 40% when user-facing workload. It also provides current and target SKU information.

You can identify underutilized machines by adjusting the CPU utilization rule on each subscription.

Resizing a virtual machine does require the machine to be shut down and restarted. There might be a period of time when it will be unavailable. Make sure you carefully time this action for minimal business impact.

Shut down the under utilized instances

Use the **Start/stop VMs during off-hours** feature of virtual machines to minimize waste. There are many configuration options to schedule start the stop times. The feature is suitable as a low-cost automation option. For information, see Start/stop VMs during off-hours solution in Azure Automation.

Azure Advisor evaluates virtual machines based on CPU and network utilization over a time period. Then, the recommended actions are shut down or resize instances and cost saving with both actions.

Spot VMs

Some workloads don't have a business requirement to complete a job within a period.

Can the workload be interrupted?

Spot VMs are ideal for workloads that can be interrupted, such as highly parallel batch processing jobs. These VMs take advantage of the surplus capacity in Azure at a lower cost. They're also well suited for experimental, development, and testing of large-scale solutions.

For more information, see Use Spot VMs in Azure.

Reserved VMs

Virtual machines are eligible for Azure Reservations. You can prepay for VM instances if you can commit to one or three years. Reserved instances are appropriate for workloads that have a long-term usage pattern.

The discount only applies to compute and not the other meters used to measure usage for VMs. The discount can be extended to other services that emit VM usage, such as Virtual machine scale sets and Container services, to name a few. For more information, see Software costs not included with Azure Reserved VM Instances and Services that get VM reservation discounts.

With reserved instances, you need to determine the VM size to buy. Analyze usage data using **Reservations Consumption APIs** and follow the recommendations of Azure portal and Azure Advisor to determine the size.

Reservations also apply to dedicated hosts. The discount is applied to all running hosts that match the reservation scope and attributes. An important consideration is the SKU for the host. When selecting a SKU, choose the VM series and type eligible to be a dedicated host. For more information, see Azure Dedicated Hosts pricing.

For information about discounts on virtual machines, see How the Azure reservation discount is applied to virtual machines.

Caching data

11/19/2021 • 2 minutes to read • Edit Online

Caching is a strategy where you store a copy of the data in front of the main data store. The cache store is typically located closer to the consuming client than the main store. Advantages of caching include faster response times and the ability to serve data quickly. In doing so, you can save on the overall cost. Be sure to assess the built-in caching features of Azure services used in your architecture. Azure also offers caching services, such as Azure Cache for Redis or Azure CDN.

For information about what type of data is suitable for caching, see Caching.

Lower costs associated with reliability and latency

Caching can be a cheaper way of storing data, providing reliability, and reducing network latency.

- Depending on the type of the data, determine if you need the complex capabilities of the backend data store, such as data consistency. If the data is fully static, you can choose to store it only in a caching store. If the data doesn't change frequently, consider placing a copy of the data in a caching store and refresh it from time to time. For example, an application stores images in blob storage. Every time it requests an image, the business logic generates a thumbnail from the main image and returns it to the caller. If the main image doesn't change too often, then return the previously generated thumbnails stored in a cache. This way you can save on resources required to process the image and lower the request response rate.
- If the backend is unavailable, the cache continues to handle requests by using the copy until the backend fails over to the backup data store.
- Caching can also be an effective way of reducing network latency. Instead of reaching the central server, the response is sent by using the cache. That way, the client can receive the response almost instantaneously. If you need higher network performance and the ability to support more client connections, choose a higher tier of the caching service. However, higher tiers will incur more costs.

Caching can be expensive

Incorrect use of caching can result in severe business outcomes and higher costs.

- Adding a cache will lead to multiple data sources in your architecture. There are added costs to keeping them in sync. You may need to fill the cache before putting it in production. Filling the cache on the application's first access can introduce latency or seeding the cache can have impact on application's start time. If you don't refresh the cache, your customers can get stale data.

  Invalidate the cache at the right time when there's latest information in the source system. Use strategies to age out the cache when appropriate.

- To make sure the caching layer is working optimally, add instrumentation. That feature will add complexity and implementation cost.

Caching services such as Azure Cache for Redis are billed on the tier you choose. Pricing is primarily determined on the cache size and network performance and they're dependent. A smaller cache will increase latency. Before choosing a tier, estimate a baseline. An approach can be by load testing the number of users and cache size.

Tradeoffs for cost

11/19/2021 • 3 minutes to read • Edit Online

As you design the workload, consider tradeoffs between cost optimization and other aspects of the design, such as security, scalability, resilience, and operability.

What is most important for the business: lowest cost, no downtime, high throughput?

An optimal design doesn't equate to a low-cost design. There might be risky choices made in favor of a cheaper solution.

Cost vs reliability

Cost has a direct correlation with reliability.

Does the cost of high availability components exceed the acceptable downtime?

Overall Service Level Agreement (SLA), Recovery Time Objective (RTO), and Recovery Point Objective (RPO) may lead to expensive design choices. If your service SLAs, RTOs, and RPOs times are short, then higher investment is inevitable for high availability and disaster recovery options.

For example, to support high availability, you choose to host the application across regions. This choice is costlier than single region because of the replication costs or the need provisioning extra nodes. Data transfer between regions will also add cost.

If the cost of high availability exceeds the cost of downtime, you can save by using Azure platform-managed replication and recover data from the backup storage.

For resiliency, availability, and reliability considerations, see the Reliability pillar.

Cost vs performance efficiency

Boosting performance will lead to higher cost.

Many factors impact performance.

Fixed or consumption-based provisioning. Avoid cost estimation of a workload at consistently high utilization. Consumption-based pricing will be more expensive that the equivalent provisioned pricing. Smooth out the peaks to get a consistent flow of compute and data. Ideally, use manual and autoscaling to find the right balance. Scaling up is more expensive than scaling out.

Azure regions. Cost scales directly with number of regions. Locating resources in cheaper regions shouldn't negate the cost of network ingress and egress or by degraded application performance because of increased latency.

Caching. Every render cycle of a payload consumes both compute and memory. You can use caching to reduce load on servers and save with pre-canned storage and bandwidth costs, and the savings can be dramatic, especially for static content services.

While caching can reduce cost, there are some performance tradeoffs. For example, Azure Traffic Manager pricing is based on the number of DNS queries that reach the service. You can reduce that number through caching and configure how often the cache is refreshed. Relying on the cache that isn't frequently updated will cause longer user failover times if an endpoint is unavailable.

Batch or real-time processing. Using dedicated resources for batch processing long running jobs will increase the cost. You can lower cost by provisioning Spot VMs but be prepared for the job to be interrupted every time Azure evicts the VM.

For performance considerations, see the Performance Efficiency pillar.

Cost vs security

Increasing security of the workload will increase cost.

As a rule, don't compromise on security. For certain workloads, you can't avoid security costs. For example, for specific security and compliance requirements, deploying to differentiated regions will be more expensive. Premium security features can also increase the cost. There are areas you can reduce cost by using native security features. For example, avoid implementing custom roles if you can use built-in roles.

For security considerations, see the Security Pillar.

Cost vs operational excellence

Investing in systems monitoring and automation might increase the cost initially but over time will reduce cost.

- IT operations processes like user or application access provisioning, incident response, and disaster recovery should be integrated with the workload.
- Cost of maintaining infrastructure is more expensive. With PaaS or SaaS services, infrastructure, platform management services, and additional operational efficiencies are included in the service pricing.

For operational considerations, see the Operational Excellence pillar.

Overview of the operational excellence pillar

11/19/2021 • 2 minutes to read • Edit Online

The operational excellence pillar covers the operations processes that keep an application running in production. Deployments must be reliable and predictable. Automated deployments reduce the chance of human error. Fast and routine deployment processes won't slow down the release of new features or bug fixes. Equally important, you must be able to quickly roll back or roll forward if an update has problems.

To assess your workload using the tenets found in the Microsoft Azure Well-Architected Framework, reference the Microsoft Azure Well-Architected Review.

We recommend the following video to help you achieve operational excellence with the Azure Well-Architected Framework:

Topics

The Microsoft Azure Well-Architected Framework includes the following topics in the operational excellence pillar:

OPERATIONAL EXCELLENCE TOPICS	DESCRIPTION
Application design	Provides guidance on how to design, build, and orchestrate workloads with DevOps principles in mind.
Monitoring	Something that enterprises have been doing for years, enriched with specifics for applications running in the cloud.
Application performance management	The monitoring and management of performance, and availability of software applications through DevOps.
Code deployment	How you deploy your application code is one of the key factors that determines your application stability.
Infrastructure provisioning	Frequently known as *Automation* or *Infrastructure as code*, this discipline refers to best practices for deploying the platform where your application will run.
Testing	Testing is fundamental to prepare for the unexpected and to catch mistakes before they impact users.

Monitoring and diagnostics are crucial. Cloud applications run in a remote data-center where you don't have full control of the infrastructure or, in some cases, the operating system. In a large application, it's not practical to log into virtual machines (VMs) to troubleshoot an issue or sift through log files. With PaaS services, there may not be a dedicated VM to log into. Monitoring and diagnostics give insight into the system, so that you know when and where failures occur. All systems must be observable. Use a common and consistent logging schema that lets you correlate events across systems.

The monitoring and diagnostics process has several distinct phases:

- *Instrumentation*: Generating the raw data from:

    - application logs
    - web server logs
    - diagnostics built into the Azure platform, and other sources.
- *Collection and storage*: Consolidating the data into one place.
- *Analysis and diagnosis*: To troubleshoot issues and see the overall health.
- *Visualization and alerts*: Using telemetry data to spot trends or alert the operations team.

Enforcing resource-level rules through Azure Policy helps ensure adoption of operational excellence best practices for all the assets, which support your workload. For example, Azure Policy can help ensure all the VMs supporting your workload adhere to a pre-approved list of VM SKUs. Azure Advisor provides a set of Azure Policy recommendations to help you quickly identify opportunities to implement Azure Policy best practices for your workload.

Use the DevOps checklist to review your design from a management and DevOps standpoint.

Next steps

Reference the operational excellence principles to guide you in your overall strategy.

Principles

Operational excellence principles

11/19/2021 • 3 minutes to read • Edit Online

Considering and improving how software is developed, deployed, operated, and maintained is one part of achieving a higher competency in operations. Equally important is providing a team culture of experimentation and growth, solutions for rationalizing the current state of operations, and incident response plans. The principles of operational excellence are a series of considerations that can help achieve excellent operational practices.

To assess your workload using the tenets found in the Azure Well-Architected Framework, see the Microsoft Azure Well-Architected Review.

DevOps methodologies

The contraction of "Dev" and "Ops" refers to replacing siloed Development and Operations to create multidisciplinary teams that now work together with shared and efficient practices and tools. Essential DevOps practices include agile planning, continuous integration, continuous delivery, and monitoring of applications.

Separation of roles

A DevOps model positions the responsibility of operations with developers. Still, many organizations do not fully embrace DevOps and maintain some degree of team separation between operations and development, either to enforce clear segregation of duties for regulated environments or to share operations as a business function.

Team collaboration

It is essential to understand if developers are responsible for production deployments end-to-end, or if a handover point exists where responsibility is passed to an alternative operations team, potentially to ensure strict segregation of duties such as the Sarbanes-Oxley Act where developers cannot touch financial reporting systems.

It is crucial to understand how operations and development teams collaborate to address operational issues and what processes exist to support and structure this collaboration. Moreover, mitigating issues might require various teams outside of development or operations, such as networking and external parties. The processes to support this collaboration should also be understood.

Workload isolation

The goal of workload isolation is to associate an application's specific resources to a team to independently manage all aspects of those resources.

Operational lifecycles

Reviewing operational incidents where the response and remediation to issues either failed or could have been optimized is vital to improving overall operational effectiveness. Failures provide a valuable learning opportunity, and in some cases, these learnings can also be shared across the entire organization. Finally, Operational procedures should be updated based on outcomes from frequent testing.

Operational metadata

Azure Tags provide the ability to associate critical metadata as a name-value pair, such as billing information (e.g., cost center code), environment information (e.g., environment type), with Azure resources, resource groups, and subscriptions. See Tagging Strategies for best practices.

Optimize build and release processes

From provisioning with Infrastructure as Code, building and releasing with CI/CD pipelines, automated testing, and embracing software engineering disciplines across your entire environment. This approach ensures the creation and management of environments throughout the software development lifecycle is consistent, repeatable, and enables early detection of issues.

Monitor the entire system and understand operational health

Implement systems and processes to monitor build and release processes, infrastructure health, and application health. Telemetry is critical to understanding the health of a workload and whether the service is meeting the business goals.

Rehearse recovery and practice failure

Run DR drills on a regular cadence and use engineering practices to identify and remediate weak points in application reliability. Regular rehearsal of failure will validate the effectiveness of recovery processes and ensure teams are familiar with their responsibilities.

Embrace operational improvement

Continuously evaluate and refine operational procedures and tasks while striving to reduce complexity and ambiguity. This approach enables an organization to evolve processes over time, optimizing inefficiencies, and learning from failures.

Use loosely coupled architecture

Enable teams to independently test, deploy, and update their systems on demand without depending on other teams for support, services, resources, or approvals.

Incident management

When incidents occur, have well thought out plans and solutions for incident management, incident communication, and feedback loops. Take the lessons learned from each incident and build telemetry and monitoring elements to prevent future occurrences.

Automation overview: Goals, best practices, and types

11/19/2021 • 4 minutes to read • Edit Online

Automation has revolutionized how businesses operate and this trend continues to evolve. Businesses have moved to automating manual processes so that engineers can focus attention on tasks that add business value. Automating business solutions allow you to:

- Activate resources on demand.
- Deploy solutions rapidly.
- Minimize human error in configuring repetitive tasks.
- Produce consistent and repeatable results.

To learn more, see Deployment considerations for automation.

Goals of automation

When automating technical processes and configurations, a common approach for some organizations is to automate what they can and leave the more difficult processes for humans to perform manually.

A goal of automation is to make tools that do what humans can do, only better. For example, a human can perform any given task once. But when the task requires repetitive runs, especially over long time periods, an automated system is better equipped to do this with more predictable results that are error-free. Increasing speed is another goal in automation. When you practice these automation goals, you can build systems that are faster, repetitive, and can run on a daily basis.

Automate toil to improve efficiency

Most automation involves a percentage of *toil*. Toil is the operational work that is related to a process that is manual, repetitive, can be automated, and has minimal value. It is counterproductive to automation but in many organizations, a small amount of toil is unavoidable. It becomes an issue when too much toil slows progress. A project's production velocity will decrease if engineers are continuously interrupted by manual tasks attributed to toil, either planned or unplanned. Too much toil can impact job satisfaction. Engineers become dissatisfied when they find themselves spending too much time on operational toil rather than on other projects.

Automation should be developed, and increased, so engineers can eliminate future toil. By reducing toil, engineers can concentrate on innovating business solutions.

For more information, see Toil automation.

Automation best practices

- **Ensure consistency:** The more manual processes are involved, the more prone to human error. This can lead to mistakes, oversights, reduction of data quality, and reliability problems.
- **Centralize mistakes:** Choose a platform to allow you to fix bugs in one place in order to fix them everywhere. This eliminates the need for communicating with other humans which increases chance of error and the possibility of the bug being re-introduced.
- **Identify issues quickly:** Complex issue may not always be able to be identified in a timely manner. However, with good automation, detection of these issues should occur quickly.
- **Maximize employee productivity:** Automation leads to more innovative solutions, additional automation,

and in general provide more value to the business. This in turn raises morale and job satisfaction. Automation does not replace jobs with technology. For example, once a process is automated, training and maintenance can be greatly reduced or eliminated. This frees engineers to spend less time on manual processes and more time on automating business solutions.

Types of automation

Three types of automation are described in this article: infrastructure deployment, infrastructure configuration, and operational tasks. These categories share the same goals and best practices mentioned previously. They differ in areas where Azure provides solutions that help achieve optimal automation. Other types of automation such as continuous deployment and continuous delivery are described further in the Operational Excellence pillar.

Infrastructure deployment

As businesses move to the cloud, they need to *repeatedly* deploy their solutions and know that their infrastructure is in a *reliable state*. To meet these challenges, you can automate deployments using a practice referred to as infrastructure as code. In code, you define the infrastructure that needs to be deployed.

Although there are many deployment technologies you can use with Azure, this article describes two of the more popular technologies:

- Azure Resource Manager (ARM) templates
- Terraform

These technologies use a *declarative* approach. This lets you state what you intend to deploy without having to write the sequence of programming commands to create it. You can deploy not only virtual machines, but also the network infrastructure, storage systems, and any other resources you may need.

Infrastructure configuration

If you don't manage configuration carefully, your business could encounter disruptions such as systems outages and security issues. Optimal configuration can enable you to quickly detect and correct configurations that could interrupt or slow performance.

Configurations such as installing software on a virtual machine, adding data to a database, or starting pods in an Azure Kubernetes Service cluster need to access Azure through an endpoint that is specific to your instance, outside of the exposed REST APIs. This enables the configuration tools to use agents, networking, or other access methods to provide resource-specific configuration options.

For example, when deploying to Azure, you may need to run post-deployment virtual machine configuration or run other arbitrary code to bootstrap the deployed Azure resources. Another example is configuration tools that can be used to configure and manage the ongoing configuration and state of Azure virtual machines.

Operational tasks

As the demand for speed in performing operational tasks increases over time, you are expected to deliver things faster and faster. The old way of manually performing operational tasks won't scale to that type of demand now. This is where automation can help. To meet on-demand delivery using an automation platform, you need to develop automation components (such as runbooks and configurations), create integrations to systems that are already in place efficiently, and operate and troubleshoot.

Advantages of automating operational tasks include:

- Optimize and extend existing processes (for example, using a PowerShell module or REST API).
- Deliver flexible and reliable services.
- Lower costs.
- Improve predictability.

Two popular options for automating operational tasks are:

- **Azure functions** - Run code without managing the underlying infrastructure on where the code is run.
- **Azure automation**- Uses a programming language to automate operational tasks in code and executed on demand.

For more information, see Automation. To see a Microsoft Ignite video, see Automating Operational and Management Tasks.

Next steps

Automate Repeatable Infrastructure

Repeatable Infrastructure

11/19/2021 • 13 minutes to read • Edit Online

Historically, deploying a new service or application involves manual work such as procuring and preparing hardware, configuring operating environments, and enabling monitoring solutions. Ideally, an organization would have multiple environments in which to test deployments. These test environments should be similar enough to production that deployment and run time issues are detected before deployment to production. This manual work takes time, is error-prone, and can produce inconsistencies between the environments if not done well.

Cloud computing changes the way we procure infrastructure. No longer are we unboxing, racking, and cabling physical infrastructure. We have internet accessible management portals and REST interfaces to help us. We can now provision virtual machines, databases, and other cloud services on demand and globally. When we no longer need cloud services, they can be easily deleted. However, cloud computing alone does not remove the effort and risk in provisioning infrastructure. When using a cloud portal to build systems, many of the same manual configuration tasks remain. Application servers require configuration, databases need networking, and firewalls need firewalling.

Azure Landing Zones (Repeatable environment configuration)

Organizations which centrally manage, govern, or support multiple workloads in the cloud will require repeatable **and consistent** environments. Azure Landing Zones provide central operations teams with a repeatable approach to environmental configuration. To deliver consistent environments, all Azure Landing Zones provide a set of common design areas, reference architecture, reference implementation, and a process to modify that deployment to fit the organization design requirements.

> **WARNING**
>
> Some organizations are following a growing industry trend towards decentralized operations (or workload operations). When operations is decentralized, the organization chooses to accept duplication of resources and potential inconsistencies in environmental configuration, in favor of reduced dependencies and full control of the environment through Azure Pipelines. Organizations who are following a decentralized operating model are less likely to leverage Azure Landing Zones to create repeatable environment configurations, but will still find value in the subsequent sections of this article.

The following is a series of links from the Cloud Adoption Framework to help deploy Azure Landing Zones:

- All Azure Landing Zones adhere to a common set of design areas to guide configuration of required environment considerations including: Identity, Network topology and connectivity, Resource organization, Governance disciplines, Operations baseline, and Business continuity and disaster recovery (BCDR)
- All Azure Landing Zones can be deployed through the Azure portal, but are designed to leverage infrastructure as code to create, test, and refactor repeatable deployments of the environmental configuration.
- The Cloud Adoption Framework provides a number of Azure Landing Zone implementation options, including:
  - Start small & expand implementation using Azure Blueprints and ARM Templates
  - Enterprise-Scale implementation using Azure Policy and ARM Templates
  - CAF Terraform modules and a variety of landing zone options

To get started with Azure Landing Zones to create consistent, repeatable environment configuration see the article series on Azure Landing Zones.

If Azure Landing Zones are not a fit for your organization, you should consider the following sections of this article to manual integrate environment configuration into your Azure Pipelines.

Deploy infrastructure with code

To fully realize deployment optimization, reduce configuration effort, and automate full environments' deployment, something more is required. One option is referred to as infrastructure as code.

Infrastructure as code (IaC) is the management of infrastructure - such as networks, virtual machines, load balancers, and connection topology - in a descriptive model, using a versioning system that is similar to what is used for source code. When you are creating an application, the same source code will generate the same binary every time it is compiled. In a similar manner, an IaC model generates the same environment every time it is applied. IaC is a key DevOps practice, and it is often used in conjunction with continuous delivery.

Ultimately, IaC allows you and your team to develop and release changes faster, but with much higher confidence in your deployments.

Gain higher confidence

One of the biggest benefits of IaC is the level of confidence you can have in your deployments, and in your understanding of the infrastructure and its configuration.

Integrate with your process. If you have a process by which code changes are peer reviewed, you can use that same process for reviewing infrastructure changes. This review process can help proactively detect problematic configurations that are difficult to detect when making manual infrastructure changes.

Consistency. Following an IaC process ensures that the whole team follows a standard, well-established process. Historically, some organizations designate a single or small set of individuals responsible for deploying and configuring infrastructure. By following a fully automated process, responsibility for infrastructure deployments moves from individuals into the automation process and tooling. This move broadens the number of team members who can initiate infrastructure deployments while maintaining consistency and quality.

Automated scanning. Many types of IaC configurations can be scanned by automated tooling. One such type of tooling is linting to check for errors in the code. Another type will scan the proposed changes to your Azure infrastructure to ensure they follow security and performance best practices, for example, ensuring that storage accounts are configured to block unsecured connections. This can be an important part of a Continuous Security approach.

Secret management. Most solutions require secrets to be maintained and managed. These include connection strings, API keys, client secrets, and certificates. Following an IaC approach means that you need to adopt best practices for managing secrets. For example, Azure Key Vault is used to store secrets securely. Key Vault can be integrated with many IaC tools and configurations to ensure that the person conducting the deployment doesn't need access to production secrets. Using a secret store with your infrastructure deployments ensures that you are adhering to the security principle of least privilege.

Access control. A fully automated IaC deployment pipeline means that an automated process should perform all changes to your infrastructure resources. This approach has many security benefits. By automating your deployments, you can be confident that changes deployed to your environment have followed the correct procedure, and you can even consider expanding the number of people who can initiate a deployment since the deployment itself is done in a fully automated way. Ideally, you would remove the ability for humans to manually modify your cloud resources and instead rely on the automated process. In case of emergencies, you may allow for this to be overridden, by using a 'break glass' account or Privileged Identity Management.

Avoid configuration drift. When using IaC, you can redeploy all of your environment templates on every

release of your solution. IaC tooling is generally built to be idempotent (i.e. to be able to be run over and over again without any bad effects). Usually, the first deployment of a IaC configuration will actually deploy the configuration, while subsequent redeployments will essentially act as 'no-ops' and have no effect. This practice helps in a few ways:

- It ensures that your IaC configurations are regularly exercised. If they are only deployed occasionally, it's much more likely they will become stale and you won't notice until it's too late. This is particularly important if you need to rely on your IaC configurations as part of a disaster recovery plan.
- It ensures that your application code and infrastructure won't get out of sync. For example, if you have an application update that needs an IaC configuration to be deployed first (such as to deploy a new database), you want to make sure you won't accidentally forget to do this in the right order. Deploying the two together in one pipeline means you are less likely to encounter these kinds of 'race conditions'.
- It helps to avoid configuration drift. If someone does accidentally make a change to a resource without following your IaC pipeline, then you want to correct this as quickly as possible and get the resource back to the correct state. By following an IaC approach, the source of truth for your environment's configuration is in code.

Manage multiple environments

Many organizations maintain multiple environments, for example, test, staging, and production. In some cases, multiple production environments are maintained for things like multi-tenanted solutions and geographically distributed applications. Ensuring consistency across these can be difficult; using infrastructure as code solutions can help.

Manage non-production environments. A common pain point for organizations is when non-production environments are dissimilar to production environments. Often, when building production and non-production environments by hand, the configuration of each will not match. This mismatch slows down the testing of changes and reduces confidence that changes will not harm a production system. When following an IaC approach, this problem is minimized. Using IaC automation, the same infrastructure configuration files can be used for all environments, producing almost identical environments. When needed, differentiation can be achieved using input parameters for each environment.

Dynamically provision environments. Once you have your IaC configurations defined, you can use them to provision new environments more efficiently. This agility can be enormously helpful when you're testing your solution. For example, you could quickly provision a duplicate of your production environment that can then be used for security penetration tests, load testing or help a developer track down a bug.

Scale production environments. Some organizations have the requirement to provision multiple production environments. For example, you may be following the Deployment Stamps pattern, or you might need to create a new instance of your environment in another geographical region. IaC configurations can be used to deploy additional instances of your solution, ensuring consistency between all environments.

Disaster recovery. In some situations, where recovery time may not be time-sensitive, IaC configurations can be used as part of a disaster recovery plan. For example, if infrastructure needs to be recreated in a second region, your IaC configurations can be used to do so. You need to consider deployment time and things like handling disaster recovery for your databases, storage accounts, and other resources that store state. All things considered, using IaC does provide an option for rapid re-creation of infrastructural assets.

When planning for disaster and recovery, ensure that your disaster recovery plans are fully tested and that they meet your recovery time requirements. This process is often referred to as a Recovery Time Objective).

Better understand your cloud resources

IaC can also help you better understand the state of your cloud resources.

Audit changes. Changes to your IaC configurations will be version-controlled in the same way as your code, such as through Git's version history. This means you can review each change that has happened, and

understand who made it and when. This can be helpful if you're trying to understand why a resource is configured a specific way.

Metadata. Many types of IaC configurations let you add metadata, like code comments, to help explain why something is done a particular way. If your organization has a culture of documenting your code, apply the same principles to your infrastructure code.

Keep everything together. It's common for a developer to work on features that require both code and infrastructure changes. By keeping infrastructure defined as code, you can group application and infrastructure code to understand the relationship between them better. For example, if you see a change to an IaC configuration on a feature branch or in a pull request, you'll have a clearer understanding of what that change relates to.

Better understand Azure itself. The Azure portal is a great way to provision and configure resources; however, it often simplifies the underlying resource model used. Using IaC will mean that you gain a much deeper understanding of what is happening in Azure and how to troubleshoot it if something isn't working correctly. For example, when creating a set of virtual machines using the Azure portal, some of the underlying resource creation is abstracted for the deployment process. When using IaC not only do you have explicit control over resource creation, very little is abstracted from the process, which provides a much richer understanding of what is deployed and how it is configured.

Categories of IaC tooling

You can use many declarative infrastructure deployment technologies with Azure. These fall into two main categories.

- **Imperative IaC** involves writing scripts in a language like Bash, PowerShell, C# script files, or Python. These programmatically execute a series of steps to create or modify your resources. When using imperative deployments, it is up to you to manage things like dependency sequencing, error control, and resource updates.
- **Declarative IaC** involves writing a definition of how you want your environment to look; the tooling then figures out how to make this happen by inspecting your current state, comparing it to the target state you've requested, and applying the differences. There's a good discussion of imperative and declarative IaC here.

There are great Azure tooling options for both models. Here we describe two of the commonly used declarative IaC technologies for Azure - ARM templates and Terraform.

Automate deployments with ARM Templates

Azure Resource Manager (ARM) Templates provide an Azure native infrastructure as code solution. ARM Templates are written in a language derived from JavaScript Object Notation (JSON), and they define the infrastructure and configurations for Azure deployments. An ARM template is declarative, you state what intend to deploy, provide configuration values, and the Azure engine takes care of making the necessary Azure REST API put requests. Additional benefits of using ARM templates for infrastructure deployments include:

- **Parallel resource deployment:** the Azure deployment engine sequences resource deployments based on defined dependencies. If dependencies do not exist between two resources, they are deployed at the same time.
- **Modular deployments:** ARM templates can be broken up into multiple template files for reusability and modularization.
- **Day one resource support:** ARM templates support all Azure resources and resource properties as they are released.
- **Extensibility:** Azure deployments can be extended using deployment scripts and other automation solutions.

- **Validation:** Azure deployments are evaluated against a validation API to catch configuration mistakes.
- **Testing:** the ARM template test toolkit provides a static code analysis framework for testing ARM templates.
- **Change preview:** ARM template what-if allows you to see what will be changed before deploying an ARM template.
- **Tooling:** Language service extensions are available for both Visual Studio Code and Visual Studio to assist in authoring ARM templates.

The following example demonstrates a simple ARM template that deploys a single Azure Storage account. In this example, a single parameter is defined to take in a name for the storage account. Under the resources section, a storage account is defined, the *storageName* parameter is used to provide a name, and the storage account details are defined. See the included documentation for an in-depth explanation of the different sections and configurations for ARM templates.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "storageName": {
            "type": "string",
            "defaultValue": "newStorageAccount"
        }
    },
    "resources": [
        {
            "name": "[parameters('storageName')]",
            "type": "Microsoft.Storage/storageAccounts",
            "apiVersion": "2019-06-01",
            "location": "[resourceGroup().location]",
            "kind": "StorageV2",
            "sku": {
                "name": "Premium_LRS",
                "tier": "Premium"
            }
        }
    ]
}
```

Learn more

- Documentation: What are ARM templates
- Learn module: Deploy consistent infrastructure with ARM Templates
- Code Samples: ARM templates

Automate deployments with Terraform

Terraform is a declarative framework for deploying and configuring infrastructure that supports many private and public clouds, Azure being one of them. It has the main advantage of offering a cloud-agnostic framework. While Terraform configurations are specific to each cloud, the framework itself is the same for all of them. Terraform configurations are written in a domain-specific language (DSL) called Hashicorp Configuration Language.

The following example demonstrates a simple Terraform configuration that deploys an Azure resource group and a single Azure Storage account.

```
resource "azurerm_resource_group" "example" {
  name     = "newStorageAccount"
  location = "eastus"
}

resource "azurerm_storage_account" "example" {
  name                     = "storageaccountname"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "GRS"
}
```

Take note, the Terraform provider for Azure is an abstraction on top of Azure APIs. This abstraction is beneficial because the API complexities are obfuscated. This abstraction comes at a cost; the Terraform provider for Azure does not always provide parity with the Azure APIs' capabilities. To learn more about using Terraform on Azure, see Using Terraform on Azure

Manual deployment

Manual deployment steps introduce significant risks where human error is concerned and also increases overall deployment times. However, in some cases, manual steps may be required. For these cases, ensure that any manual steps are documented, including roles and responsibilities.

Hotfix process

In some cases, you may have an unplanned deployment need. For instance, to deploy critical hotfixes or security remediation patches. A defined process for unplanned deployments can help prevent service availability and other deployment issues during these critical events.

Next steps

Automate infrastructure configuration

Configure infrastructure

11/19/2021 • 6 minutes to read • Edit Online

When working with Azure, many services can be created and configured programmatically using automation or infrastructure as code tooling. These tools access Azure through the exposed REST APIs or what we refer to as the Azure control plane. For example, an Azure Network Security Group can be deployed, and security group rules created using an Azure Resource Manager template. The Network Security Group and its configuration are exposed through the Azure control plane and natively accessible.

Other configurations, such as installing software on a virtual machine, adding data to a database, or starting pods in an Azure Kubernetes Service cluster cannot be accessed through the Azure control plane. These actions require a different set of configuration tools. We consider these configurations as being on the Azure data plane side, or not exposed through Azure REST APIs. These data plane enables tools to use agents, networking, or other access methods to provide resource-specific configuration options.

For example, when deploying a set of virtual machined to Azure, you may also want to install and configure a web server, stage content, and then make the content available on the internet. Furthermore, if the virtual machine configuration changes and no longer aligns with the configuration definition, you may want a configuration management system to remediate the configuration. Many options are available for these data plane configurations. This document details several and provides links for in-depth information.

Bootstrap automation

When deploying to Azure, you may need to run post-deployment virtual machine configuration or run other arbitrary code to bootstrap the deployed Azure resources. Several options are available for these bootstrapping tasks and are detailed in the following sections of this document.

Azure VM extensions

Azure virtual machine extensions are small packages that run post-deployment configuration and automation on Azure virtual machines. Several extensions are available for many different configuration tasks, such as running scripts, configuring antimalware solutions, and configuring logging solutions. These extensions can be installed and run on virtual machines using an ARM template, the Azure CLI, Azure PowerShell module, or the Azure portal. Each Azure VM has a VM Agent installed, and this agent manages the lifecycle of the extension.

A typical VM extension use case would be to use a custom script extension to install software, run commands, and perform configurations on a virtual machine or virtual machine scale set. The custom script extension uses the Azure virtual machine agent to download and execute a script. The custom script extensions can be configured to run as part of infrastructure as code deployments such that the VM is created, and then the script extension is run on the VM. Extensions can also be run outside of an Azure deployment using the Azure CLI, PowerShell module, or the Azure portal.

In the following example, the Azure CLI is used to deploy a custom script extension to an existing virtual machine, which installs a Nginx webserver.

```
az vm extension set \
  --resource-group myResourceGroup \
  --vm-name myVM --name customScript \
  --publisher Microsoft.Azure.Extensions \
  --settings '{"commandToExecute": "apt-get install -y nginx"}'
```

Learn more

Use the included code sample to deploy a virtual machine and configure a web server on that machine with the custom script extension.

- Documentation: Azure virtual machine extensions
- Code Samples: Configure VM with script extension during Azure deployment

cloud-init

cloud-init is a known industry tool for configuring Linux virtual machines on first boot. Much like the Azure custom script extension, cloud-init allows you to install packages and run commands on Linux virtual machines. cloud-init can be used for things like software installation, system configurations, and content staging. Azure includes many cloud-init enable Marketplace virtual machine images across many of the most well-known Linux distributions. For a full list, see cloud-init support for virtual machines in Azure.

To use cloud-init, create a text file named *cloud-init.txt* and enter your cloud-init configuration. In this example, the Nginx package is added to the cloud-init configuration.

```
#cloud-config
package_upgrade: true
packages:
  - nginx
```

Create a resource group for the virtual machine.

```
az group create --name myResourceGroupAutomate --location eastus
```

Create the virtual machine, specifying the *--custom-data* property with the cloud-inti configuration name.

```
az vm create \
    --resource-group myResourceGroupAutomate \
    --name myAutomatedVM \
    --image UbuntuLTS \
    --admin-username azureuser \
    --generate-ssh-keys \
    --custom-data cloud-init.txt
```

On boot, cloud-init will use the systems native package management tool to install Nginx.

Learn more

- Documentation: cloud-init support for virtual machines in Azure

Azure deployment script resource

When performing Azure deployments, you may need to run arbitrary code for bootstrapping things like managing user accounts, Kubernetes pods, or querying data from a non-Azure system. Because none of these operations are accessible through the Azure control plane, some other mechanism is required for performing this automation. To run arbitrary code with an Azure deployment, check out the `Microsoft.Resources/deploymentScripts` Azure resource.

The deployment script resource behaves similar to any other Azure resource:

- Can be used in an ARM template.
- Contain ARM template dependencies on other resources.
- Consume input, produce output.
- Use a user-assigned managed identity for authentication.

When deployed, the deployment script runs PowerShell or Azure CLI commands and scripts. Script execution

and logging can be observed in the Azure portal or with the Azure CLI and PowerShell module. Many options can be configured like environment variables for the execution environment, timeout options, and what to do with the resource after a script failure.

The following example shows an ARM template snippet with the deployment script resource configured to run a PowerShell script.

```
{
    "type": "Microsoft.Resources/deploymentScripts",
    "apiVersion": "2019-10-01-preview",
    "name": "runPowerShellScript",
    "location": "[resourceGroup().location]",
    "kind": "AzurePowerShell",
    "identity": {
        "type": "UserAssigned",
        "userAssignedIdentities": {"[parameters('identity')]": {}}
    },
    "properties": {
        "forceUpdateTag": "1",
        "azPowerShellVersion": "3.0",
        "arguments": "[concat('-sqlServer ', parameters('sqlServer'))]",
        "primaryScriptUri": "[variables('script')]",
        "timeout": "PT30M",
        "cleanupPreference": "OnSuccess",
        "retentionInterval": "P1D"
    }
}
```

Learn more

- Documentation: Use deployment scripts in templates

Configuration Management

Configuration management tools can be used to configure and manage the ongoing configuration and state of Azure virtual machines. Three popular options are Azure Automation State Configuration, Chef, and Puppet.

Azure Automation State Configuration

Azure Automation State Configuration is a configuration management solution built on top of PowerShell Desired State Configuration (DSC). State configuration works with Azure virtual machines, on-premises machines, and machines in a cloud other than Azure. Using state configuration, you can import PowerShell DSC resources and assign them to many virtual from a central location. Once each endpoint has evaluated and / or applied the desired state, state compliance is reported to Azure and can be seen on a built-in dashboard.

The following example uses PowerShell DSC to ensure the NGINX has been installed on Linux systems.

```
configuration linuxpackage {

    Import-DSCResource -Module nx

    Node "localhost" {
        nxPackage nginx {
            Name = "nginx"
            Ensure = "Present"
        }
    }
}
```

Once imported into Azure State Configuration and assigned to nodes, the state configuration dashboard provides compliance results.

Learn more

Use the included code sample to deploy Azure Automation State Configuration and several Azure virtual machines. The virtual machines are also onboarded to state configuration, and a configuration applied.

- Documentation: Get started with Azure Automation State Configuration
- Example Scenario: Azure Automation State Configuration

Chef

Chef is an automation platform that helps define how your infrastructure is configured, deployed, and managed. Additional components included Chef Habitat for application lifecycle automation rather than the infrastructure and Chef InSpec that helps automate compliance with security and policy requirements. Chef Clients are installed on target machines, with one or more central Chef Servers that store and manage the configurations.

Learn more

- Documentation: An Overview of Chef

Puppet

Puppet is an enterprise-ready automation platform that handles the application delivery and deployment process. Agents are installed on target machines to allow Puppet Master to run manifests that define the desired configuration of the Azure infrastructure and virtual machines. Puppet can integrate with other solutions such as Jenkins and GitHub for an improved DevOps workflow.

Learn more

- Documentation: How Puppet works

Next steps

Automate operational tasks

Automate operational tasks

11/19/2021 • 7 minutes to read • Edit Online

Operational tasks can include any action or activity you may perform while managing systems, system access, and processes. Some examples include rebooting servers, creating accounts, and shipping logs to a data store. These tasks may occur on a schedule, as a response to an event or monitoring alert, or ad-hock based on external factors. Like many other activities related to managing computer systems, these activities are often performed manually, which takes time, and are error-prone.

Many of these operational tasks can and should be automated. Using scripting technologies and related solutions, you can shift effort from manually performing operational tasks towards building automation for these tasks. In doing so, you achieve so much, including:

- Reduce time to perform an action.
- Reduce risk in performing the action.
- Automated response to events and alerts.
- Increased human capacity for further innovation.

When working in Azure, you have many options for automating operational tasks. This document details some of the more popular.

Azure Functions

Azure Functions allows you to run code without managing the underlying infrastructure on where the code is run. Functions provide a cost-effective, scalable, and event-driven platform for building applications and running operational tasks. Functions support running code written in C#, Java, JavaScript, Python, and PowerShell.

When creating a Function, a hosting plan is selected. Hosting plans controls how a function app is scaled, resource availability, and availability of advanced features such as virtual network connectivity and startup time. The hosting plan also influences the cost.

Functions hosting plans:

- **Consumption:** Default hosting plan, pay only for Function execution time, configurable timeout period, automatic scale.
- **Premium:** Faster start, VNet connectivity, unlimited execution duration, premium instance sizes, more predictable pricing.
- **App Service Plan:** Functions run on dedicated virtual machines and can use custom images.

For full details on consumption plans, see Azure Functions scale and hosting.

Functions provide event-driven automation; each function has a trigger associated with it. These triggers are what run the functions. Common triggers include:

- **HTTP / Webhook:** Function is run when an HTTP request is received.
- **Queue:** Function is run when a new message arrives in a message queue.
- **Blob storage:** Function is run when a new blob is created in a storage container.
- **Timer:** Function is run on a schedule.

Below are example triggers seen in the Azure portal when creating a new function

Once an event has occurred that initiates a Function, you may also want to consume data from this event or from another source. Once a Function has been completed, you may want to publish or push data to an Azure service such as a Blob Storage. Input and output are achieved using input and output bindings. For more information about triggers and bindings, see Azure Functions triggers and binding concepts.

Both PowerShell and Python are common languages for automating everyday operational tasks. Because Azure Functions supports both of these languages, it is an excellent platform for hosting, running, and auditing automated operational tasks. For example, let's assume that you would like to build a solution to facilitate self-service account creation. An Azure PowerShell Function could be used to create the account in Azure Active Directory. An HTTP trigger can be used to initiate the Function, and an input binding configured to pull the account details from the HTTP request body. The only remaining piece would be a solution that consumes the account details and creates the HTTP requests against the Function.

Learn more

- Documentation: Azure Functions PowerShell developer guide
- Documentation: Azure Functions Python developer guide

Azure Automation

PowerShell and Python are popular programming languages for automating operational tasks. Using these languages, performing operations like restarting services, moving logs between data stores, and scaling infrastructure to meet demand can be expressed in code and executed on demand. Alone, these languages do not offer a platform for centralized management, version control, or execution history. The languages also lack a native mechanism for responding to events like monitoring driven alerts. To provide these capabilities, an automation platform is needed.

Azure Automation provides an Azure-hosted platform for hosting and running PowerShell and Python code across Azure, non-Azure cloud, and on-premises environments. PowerShell and Python code is stored in an Azure Automation Runbook, which has the following attributes:

- Execute Runbooks on demand, on a schedule, or through a webhook.
- Execution history and logging.
- Integrated secrets store.

- Source Control integration.

As seen in the following image, Azure Automation provides a portal experience for managing Azure Automation Runbooks. Use the included code sample (ARM template) to deploy an Azure automation account, automation runbook, and explore Azure Automation for yourself.

Learn more

- Documentation: What is Azure Automation?

Scale operations

So far, this document has detailed options for scripting operational tasks; however, many Azure services come with built-in automation, particularly in scale operations. As application demand increases (transactions, memory consumption, and compute availability), you may need to scale the application hosting platform so that requests continue to be served. As demand decreases, scaling back not only appropriately resizes your application but also reduces operational cost.

In cloud computing, scale activities are classified into two buckets:

- **Scale-up:** Adding additional resources to an existing system to meet demand.
- **Scale-out:** Adding additional infrastructure to meet demand.

Many Azure services can be scaled up by changing the pricing tier of that service. Generally, this operation would need to be performed manually or using detection logic and custom automation.

Some Azure services support automatic scale-out, which is the focus of this section.

Azure Monitor autoscale

Azure Monitor autoscale can be used to autoscale Virtual Machine Scale Sets, Cloud Services, App Service Web Apps, and API Management service. To configure scale-out operations for these services, while in the Azure portal, select the service, and then **scale-out** under the resource settings. Select **Custom** To configure autoscaling rules. Automatic scale operations can also be configured using an Azure Resource Manager Template, the Azure PowerShell module, and the Azure CLI.

When creating the autoscale rules, configure minimum and maximum instance counts. These settings prevent inadvertent costly scale operations. Next, configure autoscale rules, at minimum one to add more instances, and one to remove instances when no longer needed. Azure Monitor autoscale rules give you fine-grain control over when a scale operation is initiated. See the Learn more section below for more information on configuring these rules.

Learn more

- Documentation: Azure Monitor autoscale overview

Azure Kubernetes Service

Azure Kubernetes Service (AKS) offers an Azure managed Kubernetes cluster. When considering scale operations in Kubernetes, there are two components:

- **Pod scaling:** Increase or decrease the number of load balanced pods to meet application demand.
- **Node scaling:** Increase or decrease the number of cluster nodes to meet cluster demand.

Azure Kubernetes Service includes automation to facilitate both of these scale operation types.

Horizontal pod autoscaler

Horizontal pod autoscaler (HPA) monitors resource demand and automatically scales pod replicas. When configuring horizontal pod autoscaling, you provide the minimum and maximum pod replicas that a cluster can run and the metrics and thresholds that initiate the scale operation. To use horizontal pod autoscaling, each pod

must be configured with resource requests and limits, and a **HorizontalPodAutoscaler** Kubernetes object must be created.

The following Kubernetes manifest demonstrates resource requests on a Kubernetes pod and also the definition of a horizontal pod autoscaler object.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-back
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
      - name: azure-vote-back
        image: redis
        # Resource requests and limits
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
        - containerPort: 6379
          name: redis
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: azure-vote-back-hpa
spec:
  maxReplicas: 10 # define max replica count
  minReplicas: 3  # define min replica count
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: azure-vote-back
  targetCPUUtilizationPercentage: 50 # target CPU utilization
```

Cluster autoscaler

Where horizontal pod autoscaling is a response to demand on a specific application of service running in a Kubernetes cluster, cluster autoscaling responds to demand on the entire Kubernetes cluster itself. If a Kubernetes cluster does not have enough compute resources or nodes to facilitate all requested pods' resource requests, some of these pods will enter a non-scheduled or pending state. In response to this situation, additional nodes can be automatically added to the cluster. Conversely, once compute resources have been freed up, the cluster nodes can automatically be removed to match steady-state demand.

Cluster autoscaler can be configured when creating an AKS cluster. The following example demonstrates this operation with the Azure CLI. This operation can also be completed with an Azure Resource Manager template.

```
az aks create \
  --resource-group myResourceGroup \
  --name myAKSCluster \
  --node-count 1 \
  --vm-set-type VirtualMachineScaleSets \
  --load-balancer-sku standard \
  --enable-cluster-autoscaler \
  --min-count 1 \
  --max-count 3
```

Cluster autoscaler can also be configured on an existing cluster using the following Azure CLI command.

```
az aks update \
  --resource-group myResourceGroup \
  --name myAKSCluster \
  --enable-cluster-autoscaler \
  --min-count 1 \
  --max-count 3
```

See the included documentation for information on fine-grain control for cluster autoscale operations.

Learn more

- Documentation: Horizontal pod autoscaling
- Documentation: AKS cluster autoscaler

Release Engineering: Application Development

11/19/2021 • 6 minutes to read • Edit Online

One of the primary goals of adopting modern release management strategies is to build systems that allow your teams to turn ideas into production delivered software with as little friction as possible. Throughout this section of the Well-Architected Framework, methods and tools for quickly and reliably delivering software are examined. You will learn about things like continuous deployment software, integration strategies, and deployment environments. Samples are provided to help you quickly get hands-on with this technology.

However, release engineering does not start with fancy deployment software, multiple deployment environments, or Kubernetes clusters. Before examining how we can quickly and reliably release software, we need to first look at how software is developed. Not only has the introduction of cloud computing had a significant impact on how software is delivered and run, but it's also had a huge downstream impact on how software is developed. For example, the introduction of container technology has changed how we can host, scale, and deprecate software. That said, containers have also impacted things like dependency management, host environment, and tooling as we develop software.

This article details many practices that you may want to consider when building strategies for developing for the cloud. Topics include:

- Development environments, or where you write your code.
- Source control and branching strategies, how you manage, collaborate on, and eventually deploy your code.

Development environments

When developing software for the cloud, or any environment, care needs to be taken to ensure that the development environment is set up for success. When setting up a development environment, you may consider questions like the following:

- How do I ensure that all dependencies are in place?
- How can I best configure my development environment to emulate a production environment?
- How do I develop code where service dependencies may exist with code already in production?

The following sections briefly detail technology that aids during the local or what is often referred to as "inner-loop" development process.

Docker Desktop

Docker Desktop is an application that provides a Docker environment on your development system. Docker Desktop includes not only the Docker runtime but application development tools and local Kubernetes environment. Using Docker Desktop, you can develop in any language, create and test container images, and when ready, push application ready container images to a container registry for production use.

Learn more

Docker Desktop

Windows Subsystem for Linux

Many applications and solutions are built on Linux. Windows Subsystem for Linux provides a Linux environment on your Windows machines, including many command-line tools, utilities, and Linux applications. Multiple GNU/Linux distributions are available and can be found in the Microsoft Store.

Learn more

Documentation: Windows Subsystem for Linux Documentation

Bridge to Kubernetes

Bridge to Kubernetes allows you to run and debug code on your development system while connected to a Kubernetes cluster. This configuration can be helpful when working on microservice type architectures. Using Bridge, you can work locally on one service, which has a dependency on other services. Rather than needing to run all dependant services on your development system or deploy your in-development code to the cluster, Bridge manages the communication between your development system and running services in your Kubernetes cluster. Essentially, the code running on your development system behaves as if it's running in the Kubernetes cluster.

Some features of Bridge:

- Works with Azure Kubernetes Service (AKS) and non-AKS clusters (in preview).
- Redirects traffic between your Kubernetes cluster and code running on your development system.
- Support for isolated traffic routing, which is important in shared clusters.
- Replicates environment variables and mounted volumes from your cluster to your development system.
- Cluster diagnostic logs are made available on your development system.

Learn more

- Documentation: Use Bridge to Kubernetes with Visual Studio Code
- Documentation: Use Bridge to Kubernetes with Visual Studio

Other tools

The tooling ecosystem for container management is rich with options. Here are a few additional tools to consider while developing container-based applications.

Podman - an open-source tool for working with containers.

Source control

Source control management (SCM) systems provide a way to control, collaborate, and peer review software changes. As software is merged into source control, the system helps manage code conflicts. Ultimately, source control provides a running history of the software, modification, and contributors. Whether a piece of software is open-sourced or private, using source control software has become a standardized method of managing software development. As detailed in later sections of the Well-Architected Framework, source control systems can also be enlightened with integrated testing, security, and release practices. As cloud practices are adopted and because so much of the cloud infrastructure is managed through code, version control systems are also becoming an integral part of infrastructure management.

Many source control systems are powered by Git. Git is a distributed version control system with related tools that allow you and your team to track source code changes during the software development lifecycle. Using Git, you can create a copy of the software, make changes, propose the changes, and receive peer review on your proposal. During peer review, Git makes it easy to see precisely the changes being proposed. Once the proposed changes have been approved, Git helps merge the changes into the source, including conflict resolution. If, at any point, the changes need to be reverted, Git can also manage rollback.

Let's examine a few aspects of version controlling software and infrastructure configurations.

Version Control and code changes

Beyond providing us with a place to store code, source control systems allow us to understand what version of the software is current and identify changes between the present and past versions. Version control solutions should also provide a method for reverting to the previous version when needed.

The following image demonstrates how Git and GitHub are used to see the proposed code changes.

Forking and Pull Requests

Using source control systems, you can create your own copies of the software source called forks. With your fork in place, changes can be made to the software without the risk that the changes will impact the software's current version. Once you are happy with the changes made in your fork, you can suggest that your changes be merged into the main source control through what is known as a pull request.

Peer Review

As updates are made to software and infrastructure configurations, version control software allows us to propose these changes before merging them into the source. During the proposal, peers can review the changes, recommend updates, and approve the changes. Source control solutions provide an excellent platform for collaboration on changes to the software.

To learn more about using Git, visit the DevOps Resource Center.

GitHub

GitHub is a popular source control system that uses Git. In addition to core Git functionality, GitHub includes several other features such as access control, collaboration features such as code issues, project boards, wikis, and an automation platform called GitHub actions.

Learn more

GitHub

Azure Repos

Azure DevOps is a collection of services for building, collaborating on, testing, and delivering software to any environment. Azure DevOps Services include Azure Repos, which is a source control system. Using Azure Repos includes unlimited free private Git repositories. Standard Git powers Azure Repos, and you can use clients and tools of your choice for working with them.

Learn more

Documentation: Azure Repos

Next steps

Release Engineering: Continuous integration

Release Engineering: Continuous integration

11/19/2021 • 4 minutes to read • Edit Online

As code is developed, updated, or even removed, having a friction-free and safe method to integrate these changes into the main code branch is paramount to enabling developers to provide value fast. As a developer, making small code changes, pushing these to a code repository, and getting almost instantaneous feedback on the quality, test coverage, and introduced bugs allows you to work faster, with more confidence, and less risk. Continuous integration (CI) is a practice where source control systems and software deployment pipelines are integrated to provide automated build, test, and feedback mechanisms for software development teams.

Continuous integration is about ensuring that software is ready for deployment but does not include the deployment itself. This article covers the basics of continuous integration and offers links and examples for more in-depth content.

Continuous integration

Continuous integration is a software development practice under which developers integrate software updates into a source control system on a regular cadence. The continuous integration process starts when an engineer creates a pull request signaling to the CI system that code changes are ready to be integrated. Ideally, integration validates the code against several baselines and tests and provides quick feedback to the requesting engineer on the status of these tests. Assuming baseline checks and testing have gone well, the integration process produces and stages assets such as compiled code and container images that will eventually deploy the updated software.

As a software engineer, continuous integration can help you deliver quality software more quickly by performing the following:

- Run automated tests against the code, providing early detection of breaking changes.
- Run code analysis to ensure code standards, quality, and configuration.
- Run compliance and security checks ensuring no known vulnerabilities.
- Run acceptance or functional tested to ensure that the software operates as expected.
- To provide quick feedback on detected issues.
- Where applicable, produce deployable assets or packages that include the updated code.

To achieve continuous integration, use software solutions to manage, integrate, and automate the process. A common practice is to use a continuous integration pipeline, detailed in this article's next section.

Continuous integration pipelines

A continuous integration pipeline involves a piece of software, in many cases, cloud-hosted, that provides a platform for running automated tests, compliance scans, reporting, and all additional components that make up the continuous integration process. In most cases, the pipeline software is attached to source control such that when pull requests are created or software is merged into a specific branch, and the continuous integration pipeline is run. Source control integration also provides the opportunity for providing CI feedback directly on the pull request.

Many solutions provide continuous integration pipeline capabilities. This article touches on both Azure Pipelines and GitHub actions and provides links to find more information.

Learn more

To learn how to create a continuous integration pipeline, see these articles:

- Azure DevOps: Create your first pipeline
- GitHub Actions: Setting up continuous integration using workflow templates

Source Control Integration

The integration of your continuous integration pipeline with your source control system is key to enabling fast, self-service, and friction-free code contributions. Source control integration enables these things:

- As pull requests are created, the CI pipeline is run, including all tests, security assessments, and other checks.
- CI test results are provided to the pull request initiator directly in the pull request, allowing for almost real-time feedback on quality.
- Another popular practice is building small reports or badges that can be presented in source control to make visible the current builds states

The following image shows the integration between GitHub and an Azure DevOps pipeline. In this example, a pull request has been created, which in turn has triggered an Azure DevOps pipeline. The pipeline status can be seen directly in the pull request.

Test Integration

A key element of continuous integration is the continual building and testing of code as code contributions are created. Testing pull requests as they are created gives quick feedback that the commit has not introduced breaking changes. The advantage is that the tests that are run by the continuous integration pipeline can be the same tests run during test-driven development.

The following code snippet shows a test step from an Azure DevOps pipeline. There are two actions occurring:

- The first task is using a popular Python testing framework to run CI tests. These tests reside in source control alongside the Python code. The test results are output to a file named *test-results.xml*.
- The second task consumes the test results and publishing them to the Azure DevOps pipeline as an integrated report.

```
- script: |
    pip3 install pytest
    pytest azure-vote/azure-vote/tests/ --junitxml=junit/test-results.xml
    continueOnError: true

- task: PublishTestResults@2
    displayName: 'Publish Test Results'
    inputs:
    testResultsFormat: 'JUnit'
    testResultsFiles: '**/test-results.xml'
    failTaskOnFailedTests: true
    testRunTitle: 'Python $(python.version)'
```

The following image shows the test results as seen in the Azure DevOps portal.

Failed tests

Failed tests should temporarily block a deployment and lead to a deeper analysis of what has happened and to either a refinement of the test or an improvement of the change that caused the test to fail.

CI Result Badges

Many developers like to show that they're keeping their code quality high by displaying a status badge in their repo. The following image shows an Azure Pipelines badge as displayed on the Readme file for an open-source project on GitHub.

Learn more

To learn how to display badges in your repositories, see these articles:

- Add an Azure Pipeline status badge to your repository.
- Add a GitHub workflow status badge to your repository.

Next steps

Release Engineering: Release testing

Testing your application and Azure environment

11/19/2021 • 6 minutes to read • Edit Online

Testing is one of the fundamental components and DevOps and agile development in general. If automation gives DevOps the required speed and agility to deploy software quickly, only through extensive testing those deployments will achieve the required reliability that customers demand.

A main tenet of a DevOps practice to achieve system reliability is the "Shift Left" principle. If developing and deploying an application is a process depicted as a series of steps going from left to right, testing should not be limited to being performed at the very end of the process (at the right). It should be shifted as much to the beginning (to the left) as possible. Errors are cheaper to repair when caught early. They can be expensive or impossible to fix later in the application life cycle.

Another aspect to consider is that testing should occur on both application code as well as infrastructure code and they should be subject to the same quality controls. As described in Infrastructure as Code, the environment where applications are running should be version-controlled and deployed through the same mechanisms as application code, and hence can be tested and validated using DevOps testing paradigms too.

You can use your favorite testing tool to run your tests, including Azure Pipelines for automated testing and Azure Testing Plans for manual testing.

There are multiple stages at which tests can be performed in the life cycle of code, and each of them has some particularities that is important to understand. In this guide, you can find a summary of the different tests that you should consider while developing and deploying applications.

Automated Testing

Automating tests is the best way to make sure that they are executed. Depending on how frequently tests are performed, they are typically limited in duration and scope, as the different types of automated tests will show:

Unit Testing

Unit tests are tests typically run by each new version of code committed into version control. Unit Tests should be extensive (should cover ideally 100% of the code) and quick (typically under 30 seconds, although this number is not a rule set in stone). Unit testing could verify things like the syntax correctness of application code, Resource Manager templates or Terraform configurations, that the code is following best practices, or that they produce the expected results when provided certain inputs.

Unit tests should be applied both to application code and infrastructure code.

Smoke Testing

Smoke tests are more exhaustive than unit tests, but still not as much as integration tests. They normally run in less than 15 minutes. Still not verifying the interoperability of the different components with each other, smoke tests verify that each of them can be correctly built and offers the expected functionality and performance.

Smoke tests usually involve building the application code, and if infrastructure, possibly testing the deployment in a test environment.

Integration Testing

After making sure that the different application components operate correctly individually, integration testing has as goal determine whether they can interact with each other as they should. Integration tests usually take longer than smoke testing, and as a consequence they are sometimes executed not as frequently. For example, running integration tests every night still offers a good compromise, detecting interoperability issues between

application components no later than one day after they were introduced.

Application Manual Testing

Manual testing is much more expensive than automated testing, and as a consequence it is run much less frequently. However, manual testing is fundamental for the correct functioning of the DevOps feedback loop, to correct errors before they become too expensive to repair, or cause customer dissatisfaction.

Acceptance Testing

There are many different ways of confirming that the application is doing what it should.

- **Blue/Green deployments**: when deploying a new application version, you can deploy it in parallel to the existing one. This way you can start redirecting clients to the new version, and if everything goes well you will decommission the old version. If there is any problem with the new deployment, you can always redirect the users back to the old one.
- **Canary releases**: you can expose new functionality of your application (ideally using feature flags) to a select group of users. If users are satisfied with the new functionality, you can extend it to the rest of the user community. In this case we are talking about releasing functionality, and not necessarily about deploying a new version of the application.
- **A/B testing**: A/B testing is similar to canary release-testing, but while canary releases focus on mitigate risk, A/B testing focus on evaluating the effectiveness of two similar ways of achieving different goals. For example, if you have two versions of the layout of a certain area of your application, you could send half of your users to one, the other half to the other, and use some metrics to see which layout works better for the application goals.

An important aspect to consider is how to measure the effectiveness of new features in the application. A way to do that is through Application Insights User Behavior Analytic, with which you can determine how people are using your application. This way you can decide whether a new feature has improved your applications without bringing effects such as decreasing usability.

Certain services in Azure offer functionality that can help with this kind of tests, such as the slot functionality in the Azure App Service, that allows having two different versions of the same application running at the same time, and redirect part of the users to one or the other.

Stress tests

As other sections of this framework have explained, designing your application code and infrastructure for scalability is of paramount importance. Testing that you can increase the application load and that both the code and the infrastructure will react to it is critical, so that your environment will adapt to changing load conditions.

During these stress tests, it is critical monitoring all the components of the system to identify whether there is any bottleneck. Every component of the system not able to scale out can turn into a scale limitation, such as active/passive network components or databases. It is hence important knowing their limits so that you can mitigate their impact into the application scale. This exercise might drive you to changing some of those components for more scalable counterparts.

It is equally important verifying that after the stress test is concluded, the infrastructure scales back down to its normal condition in order to keep costs under control.

Business Continuity Drills

Certain infrastructure test scenarios can be considered under the category of acceptance testing, such as Business continuity drills. In particular Disaster Recovery scenarios are difficult to test on-premises, but the public cloud makes this kind of tests easier. Tools such as Azure Site Recovery make it possible to start an isolated copy of the primary location in a secondary environment, so that it can be verified that the applications have come up as they should.

In case there is any problem, the Disaster Recovery procedure can be optimized, and the infrastructure in the secondary environment can be deleted.

Exploratory testing

Experts explore the application in its entirety trying to find faults or suboptimal implementations of functionality. These experts could be developers, UX specialists, product owners, actual users, and other profiles. Test plans are typically not used, since testing is left to the ability of the individual tester.

Fault injection

The same concept can be applied to the infrastructure. If the application should be resilient to infrastructure failures, introducing faults in the underlying infrastructure and observing how the application behaves is fundamental for increasing the trust in your redundancy mechanisms. Shutting down infrastructure components, purposely degrading performance, or introducing faults are ways of verifying that the application is going to react as expected when these situations occur.

Most companies use a controlled way of injecting faults in the system, although if confident with the application resiliency, automated frameworks could be used. A new science has been developed around fault injection, called Chaos Engineering.

Summary

In order to deploy software quickly and reliably, testing is a fundamental component of the development and deployment life cycle. Not only application code should be tested, but infrastructure automation and resiliency should equally be put to the test, to make sure that the application is going to perform as expected in every situation.

Next steps

Release Engineering: Performance

Performance considerations for your deployment infrastructure

11/19/2021 • 3 minutes to read • Edit Online

Build status shows if your product is in a deployable state, so builds are the heartbeat of your continuous delivery system. It's important to have a build process up and running from the first day of your product development. Since builds provide such a crucial information about the status of your product, you should always strive for fast builds.

It will be hard to fix build problem if it takes longer to build, and the team will suffer from a broken window disorder. Eventually, nobody cares if the build is broken since it's always broken and it takes lot of effort to fix it.

Build times

Here are few ways you can achieve faster builds.

- **Selecting right size VMs:** Speeding up your builds starts with selecting the right size VMs. Fast machines can make the difference between hours and minutes. If your pipelines are in Azure Pipelines, then you've got a convenient option to run your jobs using a Microsoft-hosted agent. With Microsoft-hosted agents, maintenance and upgrades are taken care of for you. For more info see, Microsoft-hosted agents.
- **Build server location:** When you're building your code, a lot of bits are moved across the wire, so inputs to the builds are fetched from a source control repository and the artifact repository, such as source code, NuGet packages, etc. At the end, the output from the build process needs to be copied, not only the compiled artifacts, but also test reports, code coverage results, and debug symbols. So it is important that these copy actions are fast. If you are using your own build server, ensure that the build server is located near the sources and a target location, and it can reduce the duration of your build considerably.
- **Scaling out build servers:** A single build server may be sufficient for a small product, but as the size and the scope of the product and the number of teams working on the product increases, the single server may not be enough. Scale your infrastructure horizontally over multiple machines when you reach the limit. For more info see, how you can leverage Azure DevOps Agent Pools.
- **Optimizing the build:**
  - Add parallel build execution so we can speed up the build process. For more info see, Azure DevOps parallel jobs.
  - Enable parallel execution of test suites, which is often a huge time saver, especially when executing integration and UI tests. For more info see, Run tests in parallel using Azure Pipeline.
  - Use the notion of a multiplier, where you can scale out your builds over multiple build agents. For more info see, Organizing Azure Pipeline into Jobs.
  - Move a part of the test feedback loop to the release pipeline. This improves the build speed, and hence the speed of the build feedback loop.
  - Publish the build artifacts to the package management system solution, and hence publish to a NuGet feed at the end of a build.

Human intervention

It's important to select different builds for different purpose.

- **CI builds:** Purpose of this build is to ensure it compiles and unit tests run. This build gets triggered at each commit or set of commits over a period of time. It serves as the heartbeat of the project, provides quality feedback to the team immediately. For more info see, CI triggers or Batching CI builds.
- **Nightly build:** Purpose of this build is not only to compile but also ensure necessary integration/regression tests are run. This build can take up some more time, because we need to do some extra steps to get additional information about the product. For example, metrics about the state of the software using SonarQube. It may also contain a set of regression tests and integration tests and it may also deploy the solution to a temporary machine to verify the solution is continuing to work. For more info see, scheduling builds using cron syntax
- **Release build:** Besides compiling, running test this build additionally compiles the API documentation, compliance reports, code signing, and other steps which are not required every time the code is built. Finally this build provide the golden copy that will be pushed to the release pipeline to finally deploy in the production environment. Generally release build is gets triggered manually instead of a CI trigger.

Next steps

Release Engineering: Deployment

Deployment considerations for DevOps

11/19/2021 • 4 minutes to read • Edit Online

As you provision and update Azure resources, application code, and configuration settings, a repeatable and predictable process will help you avoid errors and downtime. We recommend automated processes for deployment that you can run on demand and rerun if something fails. After your deployment processes are running smoothly, process documentation can keep them that way.

Automation

To activate resources on demand, deploy solutions rapidly, minimize human error, and produce consistent and repeatable results, be sure to automate deployments and updates.

Automate as many processes as possible

The most reliable deployment processes are automated and *idempotent* — that is, repeatable to produce the same results.

- To automate provisioning of Azure resources, you can use Terraform, Ansible, Chef, Puppet, Azure PowerShell, Azure CLI, or Azure Resource Manager templates.
- To configure VMs, you can use cloud-init (for Linux VMs) or Azure Automation State Configuration (DSC).
- To automate application deployment, you can use Azure DevOps Services, Jenkins, or other CI/CD solutions.

As a best practice, create a repository of categorized automation scripts for quick access, documented with explanations of parameters and examples of script use. Keep this documentation in sync with your Azure deployments, and designate a primary person to manage the repository.

Automation scripts can also activate resources on demand for disaster recovery.

Automate and test deployment and maintenance tasks

Distributed applications consist of multiple parts that must work together. Deployment should take advantage of proven mechanisms, such as scripts, that can update and validate configuration and automate the deployment process. Test all processes fully to ensure that errors don't cause additional downtime.

Implement deployment security measures

All deployment tools must incorporate security restrictions to protect the deployed application. Define and enforce deployment policies carefully, and minimize the need for human intervention.

Release process

One of the challenges with automating deployment is the cut-over itself, taking software from the final stage of testing to live production. You usually need to do this quickly in order to minimize downtime. The blue-green deployment approach does this by ensuring you have two production environments, as identical as possible.

Document release process

Without detailed release process documentation, an operator might deploy a bad update or might improperly configure settings for your application. Clearly define and document your release process, and ensure that it's available to the entire operations team.

Stage your workloads

Deployment to various stages and running tests/validations at each stage before moving on to the next ensures friction free production deployment.

With good use of staging and production environments, you can push updates to the production environment in a highly controlled way and minimize disruption from unanticipated deployment issues.

- *Blue-green deployment* involves deploying an update into a production environment that's separate from the live application. After you validate the deployment, switch the traffic routing to the updated version. One way to do this is to use the staging slots available in Azure App Service to stage a deployment before moving it to production.
- *Canary releases* are similar to blue-green deployments. Instead of switching all traffic to the updated application, you route only a small portion of the traffic to the new deployment. If there's a problem, revert to the old deployment. If not, gradually route more traffic to the new version. If you're using Azure App Service, you can use the Testing in production feature to manage a canary release.

Test environments

If development and test environments don't match the production environment, it is hard to test and diagnose problems. Therefore, keep development and test environments as close to the production environment as possible. Make sure that test data is consistent with the data used in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

Logging and auditing

To capture as much version-specific information as possible, implement a robust logging strategy. If you use staged deployment techniques, more than one version of your application will be running in production. If a problem occurs, determine which version is causing it.

High availability considerations

An application that depends on a single instance of a service creates a single point of failure. To improve resiliency and scalability, provision multiple instances.

- For Azure App Service, select an App Service plan that offers multiple instances.
- For Azure Cloud Services, configure each of your roles to use multiple instances.
- For Azure Virtual Machines, ensure that your architecture includes more than one VM and that each VM is included in an availability set.

Consider deploying across multiple regions

We recommend deploying all but the least critical applications and application services across multiple regions. If your application is deployed to a single region, in the rare event that the entire region becomes unavailable, the application will also be unavailable. If you choose to deploy to a single region, consider preparing to redeploy to a secondary region as a response to an unexpected failure.

Redeploy to a secondary region

If you run applications and databases in a single, primary region with no replication, your recovery strategy might be to redeploy to another region. This solution is affordable but most appropriate for non-critical applications that can tolerate longer recovery times. If you choose this strategy, automate the redeployment process as much as possible and include redeployment scenarios in your disaster response testing.

To automate your redeployment process, consider using Azure Site Recovery.

Next steps

Release Engineering: Rollback

Release Engineering: Rollback

11/19/2021 • 2 minutes to read • Edit Online

In some cases, a new software deployment can harm or degrade the functionality of a software system. When building your solutions, it is essential to anticipate deployment issues and to architect solutions that provide mechanisms for fixing problematic deployments. Rolling back a deployment involves reverting the deployment to a known good state. Rollback can be accomplished in many different ways. Several Azure services support native mechanisms for rolling back to a previous state. Some of these services are detailed in this article.

Azure App Service

When deploying an application to Azure App Service, consider utilizing deployment slots. Deployment slots are running instances of the application, each with a separate host name. Slots can be used to stage and test applications before promoting to a production slot. A deployment slot can be created to hold the last known good instance of your application. In the event of an issue or problematic deployment, the production slot can be swapped with the known good slot to bring the application back to a known good state.

Learn more

For more information on using Azure App Service deployment slots, see Set up staging environments in Azure App Service

Azure Kubernetes Service (AKS)

A Kubernetes deployment defines the desired state for a particular workload running in the cluster. For example, a deployment may declare that a workload consists of 3 replicas of a specific pod that should be running at all times. The deployment object creates a ReplicaSet and the associate Pods. When updating a workload, the deployment itself can be revised, which will generally roll out a new container image to the deployment pods. Assuming multiple replicas of the pods exist, this rollout can happen in a controlled and staged manner such that no downtime occurs.

If a deployment introduces breaking changes or unintentional results, it can be reverted to an earlier state.

In this case, a deployment named *demorollback* contains three replicas of a pod.

```
$ kubectl get deployments

NAME           READY   UP-TO-DATE   AVAILABLE   AGE
demorollback   3/3     3            3           28s
```

As the deployment is updated, deployment history is retained.

```
$ kubectl rollout history deployment demorollback

deployment.apps/demorollback
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

If an updated deployment has introduced issues, the `kubectl rollout undo` command can be used to revert to a previous deployment revision.

```
$ kubectl rollout undo deploy demorollback --to-revision=1

deployment.apps/demorollback rolled back
  Volumes:      <none>
```

Learn more

For more information, see Kubernetes Deployments

Azure Resource Manager (ARM) deployments

A deployment record is created when deploying Azure infrastructure and solutions with Azure Resource Manager (ARM) templates. When creating a new deployment, you can provide a previously known good deployment so that if the current deployment fails, the previous know good deployment is redeployed. There are several considerations and caveats when using this functionality. See the documentation linked below for these details.

Learn more

For more information, see Rollback on an error to successful deployment

Logic apps

When making changes to an Azure logic application, a new version of the application is created. Azure maintains a history of versions and can revert or promote any previous version. To do so, in the Azure portal, select your logic app > **Versions**. Previous versions can be selected on the versions pane, and the application can be inspected both in the code view and the visual designer view. Select the version you would like to revert to, and click the **Promote** option and then **Save**.

Learn more

For more information, see Manage logic apps in the Azure portal

Monitoring for workloads

11/19/2021 • 8 minutes to read • Edit Online

Traditional application and infrastructure monitoring is based on the application running successfully and its response time. However, cloud-based monitoring offers many more opportunities you can leverage to give users the best experience. In a production environment, it's important to track the way customers use your system, trace resource utilization, and generally monitor the health and performance of your system. You can use this information as a diagnostic aid to detect and correct issues, and also to help spot potential problems, and prevent them from occurring.

Application monitoring

Application Insights is the Azure Service that allows you to:

- Verify that your application is running correctly.
- Makes application troubleshooting easier.
- Provides custom business telemetry to indicate whether your application is being used as intended.

Make sure you leverage all the rich information that Application Insights can provide about your application. This list is not exhaustive, but the following items provide visibility into what Application Insights can offer:

- Application Insights offers you a default dashboard with an educated guess of the most important metrics you will be interested in. You can then modify it and customize it to your own needs.
- By instrumenting your application correctly, Application Insights will give you performance statistics both from a client and a server perspective
- The Application Map will show you application dependencies in other services such as backend APIs or databases, allowing to determine visually where performance problems lie
- Smart Detection will warn you when anomalies in performance or utilization patterns happen
- Usage Analysis can give you telemetry on which features of your application are most frequently used, or whether all your application functionality is being used. This feature is especially useful after changes to the application functionality, to verify whether those changes were successful
- Release annotations are visual indicators in your Application Insights charts of new builds and other events, so that you can visually correlate changes in application performance to code releases, being able to quickly pinpoint performance problems.
- Cross-component transaction diagnostics allow you to follow failed transactions to find the point in the architecture where the fault was originated.
- Snapshot Debugger, to automatically collect a snapshot of a live application in case of an exception, to analyze it at a later stage.

To use Application Insights you have two options:

- Use **codeless monitoring**, where onboarding your app to Application Insights doesn't require any code change.
- Use **code-based monitoring**, where you instrument your code to send telemetry to Application Insights using the Software Development Kit for your programming language of choice.

You can certainly use other Application Performance Management tools to monitor your application on Azure, such as NewRelic or AppDynamics, but Application Insights will give you the most seamless and integrated experience.

Platform monitoring

Application Insights is actually one of the components of Azure Monitor, which gives you rich metrics and logs to verify the state of your complete Azure landscape. No matter whether your application is running on Virtual Machines, App Services, or Kubernetes, Azure Monitor will help you to follow the state of your infrastructure, and to react promptly if there are any issues.

Make sure not only to monitor your compute elements supporting your application code, but your data platform as well: databases, storage accounts, or data lakes should be closely monitored, since a low performance of the data tier of an application could have serious consequences.

Container insights

Should your application run on Azure Kubernetes Service, Azure Monitor allows you to easily monitor the state of your cluster, nodes, and pods. Easy to configure for AKS clusters, Container Insights delivers quick, visual, and actionable information: from the CPU and memory pressure of your nodes to the logs of individual Kubernetes pods.

Additionally, for operators that prefer using the open-source Kubernetes monitoring tool Prometheus but still like the ease of use of Azure Monitor Container Insights, both solutions can integrate with each other.

The Sidecar Pattern adds a separate container with responsibilities that are required by the main container. A common use case is for running logging utilities and monitoring agents.

Network monitoring

Regardless the form factor or programming language your application is based on, the network connecting your code to your users can make or break the experience that your application provides. As a consequence monitoring and troubleshooting the network can be decisive for an operations team. The component of Azure Monitor that manages the network components is called Network Watcher, a collection of network monitoring and troubleshooting tools. Some of these tools are:

- Traffic Analytics will give you an overview of the traffic in your Virtual Networks, as well as the percentage coming from malicious IP addresses, leveraging Microsoft Threat Intelligence databases. This tool will show you as well the systems in your virtual networks that generate most traffic, so that you can visually identify bottlenecks before they degenerate into problems.
- Network Performance Monitor can generate synthetic traffic to measure the performance of network connections over multiple links, giving you a perspective on the evolution of WAN and Internet connections over time, as well as offering valuable monitoring information about Microsoft ExpressRoute circuits.
- VPN diagnostics can help troubleshooting site-to-site VPN connections connecting your applications to users on-premises.
- Connection Monitor allows you to measure the network availability between sets of endpoints.

Other information sources

Not only your application components are producing data, but there are many other signals that you need to track to effectively operate a cloud environment:

- **Activity Log**: this is a trail audit that lets you see every change that has gone through Azure APIs. It can be critical to understand sudden performance changes or problems, that might have been due to a misconfiguration of the Azure platform.
- **Azure Service Health**: sometimes outages are provoked not by configuration changes, but by glitches in the Azure platform itself. You can find information about any Azure-related problem impacting your application in the Azure Service Health logs.
- **Azure Advisor**: find here recommendations about how to optimize your Azure platform to reduce costs, improve your security posture, or increase the availability of your environment.
- **Microsoft Defender for Cloud**: not a focus of this pillar, but to be included for completeness: Microsoft

Defender for Cloud can help you to understand whether your Azure resources are configured according to security best practices

Monitor external dependencies

Just as important as monitoring your applications and Azure infrastructure, monitoring external dependencies is also critical. Ensure that you have solutions to gain insight into the health and availability of external dependencies such as third party services, package repositories, and other related applications.

Azure monitor can have a role in monitoring external dependencies. Azure Monitor can ingest data from several custom data sources, which provides some extensibility into what can be monitored. Utilizing custom data sources for external dependencies provides you with the capabilities of Azure Monitor on virtual any service, endpoint, or software solution.

To learn more about ingesting data from custom sources, see the following articles.

- Collect custom logs with Log Analytics agent in Azure Monitor
- Collecting custom JSON data sources with Log Analytics agent for Linux
- Send log data to Azure Monitor with the HTTP Data Collector API

Monitoring best practices

Event correlation

One critical advantage of Azure Monitor is that it is the monitoring tool for the whole Azure platform. As the previous sections have shown, Azure Monitor holds metrics and logs relevant to your application code, the platform where it is running, the data components, as well as the network connecting the application to its users. This enables operators to compare metrics of different application components to each other, and find out dependencies that might have been hidden otherwise.

Dashboards in Azure offer a great way of exposing the rich information contained in Azure Monitor to other users. Make sure to create shared dashboards in order to expose relevant information to the different groups involved in operating your application, including Developers and Operators. If more complex visualizations are required, Azure Monitor data can be exported to Power BI for advanced data analysis.

Notifications

Whether it is for application, network or platform monitoring, you should not expect operators to constantly look at dashboards. Instead, alerts should be used to send proactive notifications to the relevant individuals that will react on them. Action groups in Azure Monitor can be used to notify multiple recipients, to trigger automated actions, or even to automatically open tickets in IT Service Management Tools such as ServiceNow.

Automation around alerts is critical due to the highly collaborative nature of DevOps and the inherent speed needed for effective incident management. Earlier this year, a report from DevOps.com came out stating that 80% of IT teams are alerted to critical incidents via email. Email is an effective form of communication, but it shouldn't be the most common notification method for a critical issue. Instead, if you can define actions to be executed upon receiving certain alerts (such as scaling up or down) your system will be self-healing.

Other monitoring tasks

Beyond Azure Monitor, you will want to keep an eye on certain events to make sure that your application is running smoothly:

- Review Azure subscription limits for your resources, and make sure you are not coming too close.
- Understand Azure support plans. Refer to Azure support FAQs. Familiarize your team with Azure support.
- Make sure that you monitor expiration dates of digital certificates, or even better, configure automatic digital certificate renewal with Azure Key Vault.

Summary

You can use any monitoring platform to manage your Azure resources. Microsoft's first party offering is Azure Monitor, a comprehensive solution for metrics and logs from the infrastructure to the application code, including the ability to trigger alerts and automated actions as well as data visualization.

Next steps

Monitoring checklist

Alerting

11/19/2021 • 4 minutes to read • Edit Online

Alerting is the process of analyzing the monitoring and instrumentation data, and generating a notification if a significant event is detected.

Alerting helps ensure that the system remains healthy, responsive, and secure. It's an important part of any system that makes performance, availability, and privacy guarantees to the users where the data might need to be acted on immediately. An operator might need to be notified of the event that triggered the alert. Alerting can also be used to invoke system functions such as autoscaling.

Alerts from tools such as Splunk or Azure Monitor proactively notify or respond to operational states that deviate from norm. Alerts can also enable cost-awareness by watching budgets, limits, and helping workload teams scale appropriately.

Alerting depends on the following instrumentation data:

- *Security events*: If the event logs indicate that repeated authentication or authorization failures are occurring, the system might be under attack and an operator should be informed.
- *Performance metrics*: The system must quickly respond if a particular performance metric exceeds a specified threshold.
- *Availability information*: If a fault is detected, it might be necessary to quickly restart one or more subsystems, or failover to a backup resource. Repeated faults in a subsystem might indicate more serious concerns.

Operators might receive alert information by using many delivery channels such as email, a pager device, or an SMS text message. An alert might also include an indication of how critical a situation is. Many alerting systems support subscriber groups, and all operators who are members of the same group can receive the same set of alerts.

An alerting system should be customizable, and the appropriate values from the underlying instrumentation data can be provided as parameters. This approach enables an operator to filter data and focus on those thresholds or combinations of interest values. Note that in some cases, the raw instrumentation data can be provided to the alerting system. In other situations, it might be more appropriate to supply aggregated data. For example, an alert can be triggered if the CPU utilization for a node has exceeded `90%` over the past `10` minutes. The details provided to the alerting system should also include any appropriate summary and context information. This data can help reduce the possibility that false-positive events will trip an alert.

Alert rules and actions

When configuring alerts in Azure, ensure the following items are configured:

- *Alert rule*: This rule includes the alert scope or set of resources on which to alert. The alert rule also includes an alert condition. The condition is a query of Azure monitor data and the data threshold on which to raise an alert.
- *Action group*: This group defines the action to take once an alert has been triggered.

An alert rule can be defined using many different data streams such as metric values, log search queries, and activity log events.

For more information on alerts, reference Overview of alerts in Microsoft Azure.

Alert prioritization

Prioritizing alerts with a specific severity or urgency helps operational teams in cases where multiple events require intervention at the same time. For example, alerts concerning critical system flows might require special attention. When creating an alert, ensure you establish and set the correct priority.

Alert owners

Having well-defined owners and response playbooks per alert is vital to optimizing operational effectiveness. Alerts don't have to be only technical. For example, the budget owner should be made aware of capacity issues so that budgets can be adjusted and discussed.

Alert response

Each Azure Monitor alert can be configured with one or more associated action groups. An action group can respond to an Azure Monitor alert in the following ways:

- Send an email, SMS message, or push notification
- Execute an Azure Function
- Execute an Azure Automation runbook
- Execute an Azure Logic App
- Send a request to a webhook endpoint

By using an Azure Monitor action group, you can raise a notification when an alert is created and respond with automated action.

For more information on using Azure Monitor action groups, reference Create and manage action groups.

Alert dashboarding

The default Alerts page provides a summary of alerts that are created within a particular time range. It displays the total alerts for each severity, with columns that identify the total number of alerts in each state for each severity.

In addition to the default Azure Monitor alert dashboard, custom dashboards can be created using log analytics data. This dashboard can be tailored to reflect the current and past start of all alerts.

For more information on creating dashboards, reference Create and share dashboards of Log Analytics data.

Alert integrations

Because of the flexibility provided with Azure Monitor, alerts, and action groups, integration possibilities are essentially limitless. For example, if you have a custom solution that ingests data through an incoming API, this can be engaged with an Azure Monitor action group each time an alert is raised. This flexibility allows for integration with a custom system, ITSM solutions, and other work tracking solutions. Many partner integrations are ready to use out of the box.

For information on Azure Monitor and ITSM integration, reference IT Service Management Connector Overview.

Next steps

Return to the operational excellence overview.

Operational Excellence Overview

DevOps Checklist

11/19/2021 • 14 minutes to read • Edit Online

DevOps is the integration of development, quality assurance, and IT operations into a unified culture and set of processes for delivering software. Use this checklist as a starting point to assess your DevOps culture and process.

Culture

Ensure business alignment across organizations and teams. Conflicts over resources, purpose, goals, and priorities within an organization can be a risk to successful operations. Ensure that the business, development, and operations teams are all aligned.

Ensure the entire team understands the software lifecycle. Your team needs to understand the overall lifecycle of the application, and which part of the lifecycle the application is currently in. This helps all team members know what they should be doing now, and what they should be planning and preparing for in the future.

Reduce cycle time. Aim to minimize the time it takes to move from ideas to usable developed software. Limit the size and scope of individual releases to keep the test burden low. Automate the build, test, configuration, and deployment processes whenever possible. Clear any obstacles to communication among developers, and between developers and operations.

Review and improve processes. Your processes and procedures, both automated and manual, are never final. Set up regular reviews of current workflows, procedures, and documentation, with a goal of continual improvement.

Do proactive planning. Proactively plan for failure. Have processes in place to quickly identify issues when they occur, escalate to the correct team members to fix, and confirm resolution.

Learn from failures. Failures are inevitable, but it's important to learn from failures to avoid repeating them. If an operational failure occurs, triage the issue, document the cause and solution, and share any lessons that were learned. Whenever possible, update your build processes to automatically detect that kind of failure in the future.

Optimize for speed and collect data. Every planned improvement is a hypothesis. Work in the smallest increments possible. Treat new ideas as experiments. Instrument the experiments so that you can collect production data to assess their effectiveness. Be prepared to fail fast if the hypothesis is wrong.

Allow time for learning. Both failures and successes provide good opportunities for learning. Before moving on to new projects, allow enough time to gather the important lessons, and make sure those lessons are absorbed by your team. Also give the team the time to build skills, experiment, and learn about new tools and techniques.

Document operations. Document all tools, processes, and automated tasks with the same level of quality as your product code. Document the current design and architecture of any systems you support, along with recovery processes and other maintenance procedures. Focus on the steps you actually perform, not theoretically optimal processes. Regularly review and update the documentation. For code, make sure that meaningful comments are included, especially in public APIs, and use tools to automatically generate code documentation whenever possible.

Share knowledge. Documentation is only useful if people know that it exists and can find it. Ensure the documentation is organized and easily discoverable. Be creative: Use brown bags (informal presentations),

videos, or newsletters to share knowledge.

Development

Provide developers with production-like environments. If development and test environments don't match the production environment, it is hard to test and diagnose problems. Therefore, keep development and test environments as close to the production environment as possible. Make sure that test data is consistent with the data used in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

Ensure that all authorized team members can provision infrastructure and deploy the application. Setting up production-like resources and deploying the application should not involve complicated manual tasks or detailed technical knowledge of the system. Anyone with the right permissions should be able to create or deploy production-like resources without going to the operations team.

> This recommendation doesn't imply that anyone can push live updates to the production deployment. It's about reducing friction for the development and QA teams to create production-like environments.

Instrument the application for insight. To understand the health of your application, you need to know how it's performing and whether it's experiencing any errors or problems. Always include instrumentation as a design requirement, and build the instrumentation into the application from the start. Instrumentation must include event logging for root cause analysis, but also telemetry and metrics to monitor the overall health and usage of the application.

Track your technical debt. In many projects, release schedules can get prioritized over code quality to one degree or another. Always keep track when this occurs. Document any shortcuts or other suboptimal implementations, and schedule time in the future to revisit these issues.

Consider pushing updates directly to production. To reduce the overall release cycle time, consider pushing properly tested code commits directly to production. Use feature toggles to control which features are enabled. This allows you to move from development to release quickly, using the toggles to enable or disable features. Toggles are also useful when performing tests such as canary releases, where a particular feature is deployed to a subset of the production environment.

Testing

Automate testing. Manually testing software is tedious and susceptible to error. Automate common testing tasks and integrate the tests into your build processes. Automated testing ensures consistent test coverage and reproducibility. Integrated UI tests should also be performed by an automated tool. Azure offers development and test resources that can help you configure and execute testing. For more information, see Development and test.

Test for failures. If a system can't connect to a service, how does it respond? Can it recover once the service is available again? Make fault injection testing a standard part of review on test and staging environments. When your test process and practices are mature, consider running these tests in production.

Test in production. The release process doesn't end with deployment to production. Have tests in place to ensure that deployed code works as expected. For deployments that are infrequently updated, schedule production testing as a regular part of maintenance.

Automate performance testing to identify performance issues early. The impact of a serious performance issue can be as severe as a bug in the code. While automated functional tests can prevent application bugs, they might not detect performance problems. Define acceptable performance goals for metrics like latency, load times, and resource usage. Include automated performance tests in your release pipeline, to make sure the application meets those goals.

Perform capacity testing. An application might work fine under test conditions, and then have problems in production due to scale or resource limitations. Always define the maximum expected capacity and usage limits. Test to make sure the application can handle those limits, but also test what happens when those limits are exceeded. Capacity testing should be performed at regular intervals.

After the initial release, you should run performance and capacity tests whenever updates are made to production code. Use historical data to fine-tune tests and to determine what types of tests need to be performed.

Perform automated security penetration testing. Ensuring your application is secure is as important as testing any other functionality. Make automated penetration testing a standard part of the build and deployment process. Schedule regular security tests and vulnerability scanning on deployed applications, monitoring for open ports, endpoints, and attacks. Automated testing does not remove the need for in-depth security reviews at regular intervals.

Perform automated business continuity testing. Develop tests for large-scale business continuity, including backup recovery and failover. Set up automated processes to perform these tests regularly.

Release

Automate deployments. Automate deploying the application to test, staging, and production environments. Automation enables faster and more reliable deployments, and ensures consistent deployments to any supported environment. It removes the risk of human error caused by manual deployments. It also makes it easy to schedule releases for convenient times, to minimize any effects of potential downtime. Have systems in place to detect any problems during rollout, and have an automated way to roll forward fixes or roll back changes.

Use continuous integration. Continuous integration (CI) is the practice of merging all developer code into a central codebase on a regular schedule, and then automatically performing standard build and test processes. CI ensures that an entire team can work on a codebase at the same time without having conflicts. It also ensures that code defects are found as early as possible. Preferably, the CI process should run every time that code is committed or checked in. At the very least, it should run once per day.

> Consider adopting a trunk based development model. In this model, developers commit to a single branch (the trunk). There is a requirement that commits never break the build. This model facilitates CI, because all feature work is done in the trunk, and any merge conflicts are resolved when the commit happens.

Consider using continuous delivery. Continuous delivery (CD) is the practice of ensuring that code is always ready to deploy, by automatically building, testing, and deploying code to production-like environments. Adding continuous delivery to create a full CI/CD pipeline will help you detect code defects as soon as possible, and ensures that properly tested updates can be released in a very short time.

> Continuous *deployment* is an additional process that automatically takes any updates that have passed through the CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning, and may not be appropriate for all teams.

Make small incremental changes. Large code changes have a greater potential to introduce bugs. Whenever possible, keep changes small. This limits the potential effects of each change, and makes it easier to understand and debug any issues.

Control exposure to changes. Make sure you're in control of when updates are visible to your end users. Consider using feature toggles to control when features are enabled for end users.

Implement release management strategies to reduce deployment risk. Deploying an application update to production always entails some risk. To minimize this risk, use strategies such as canary releases or

blue-green deployments to deploy updates to a subset of users. Confirm the update works as expected, and then roll the update out to the rest of the system.

Document all changes. Minor updates and configuration changes can be a source of confusion and versioning conflict. Always keep a clear record of any changes, no matter how small. Log everything that changes, including patches applied, policy changes, and configuration changes. (Don't include sensitive data in these logs. For example, log that a credential was updated, and who made the change, but don't record the updated credentials.) The record of the changes should be visible to the entire team.

Consider making infrastructure immutable. Immutable infrastructure is the principle that you shouldn't modify infrastructure after it's deployed to production. Otherwise, you can get into a state where ad hoc changes have been applied, making it hard to know exactly what changed. Immutable infrastructure works by replacing entire servers as part of any new deployment. This allows the code and the hosting environment to be tested and deployed as a block. Once deployed, infrastructure components aren't modified until the next build and deploy cycle.

Monitoring

Make systems observable. The operations team should always have clear visibility into the health and status of a system or service. Set up external health endpoints to monitor status, and ensure that applications are coded to instrument the operations metrics. Use a common and consistent schema that helps you correlate events across systems. Azure Diagnostics and Application Insights are the standard method of tracking the health and status of Azure resources. Azure Monitor also provides centralized monitoring and management for cloud or hybrid solutions.

Aggregate and correlate logs and metrics. A properly instrumented telemetry system will provide a large amount of raw performance data and event logs. Make sure that telemetry and log data is processed and correlated in a short period of time, so that operations staff always have an up-to-date picture of system health. Organize and display data in ways that give a cohesive view of any issues, so that whenever possible it's clear when events are related to one another.

> Consult your corporate retention policy for requirements on how data is processed and how long it should be stored.

Implement automated alerts and notifications. Set up monitoring tools like Azure Monitor to detect patterns or conditions that indicate potential or current issues, and send alerts to the team members who can address the issues. Tune the alerts to avoid false positives.

Monitor assets and resources for expirations. Some resources and assets, such as certificates, expire after a given amount of time. Make sure to track which assets expire, when they expire, and what services or features depend on them. Use automated processes to monitor these assets. Notify the operations team before an asset expires, and escalate if expiration threatens to disrupt the application.

Management

Automate operations tasks. Manually handling repetitive operations processes is error-prone. Automate these tasks whenever possible to ensure consistent execution and quality. Code that implements the automation should be versioned in source control. As with any other code, automation tools must be tested.

Take an infrastructure-as-code approach to provisioning. Minimize the amount of manual configuration needed to provision resources. Instead, use scripts and Azure Resource Manager templates. Keep the scripts and templates in source control, like any other code you maintain.

Consider using containers. Containers provide a standard package-based interface for deploying applications. Using containers, an application is deployed using self-contained packages that include any

software, dependencies, and files needed to run the application, which greatly simplifies the deployment process.

Containers also create an abstraction layer between the application and the underlying operating system, which provides consistency across environments. This abstraction can also isolate a container from other processes or applications running on a host.

Implement resiliency and self-healing. Resiliency is the ability of an application to recover from failures. Strategies for resiliency include retrying transient failures, and failing over to a secondary instance or even another region. For more information, see Designing reliable Azure applications . Instrument your applications so that issues are reported immediately and you can manage outages or other system failures.

Have an operations manual. An operations manual or *runbook* documents the procedures and management information needed for operations staff to maintain a system. Also document any operations scenarios and mitigation plans that might come into play during a failure or other disruption to your service. Create this documentation during the development process, and keep it up to date afterwards. This is a living document, and should be reviewed, tested, and improved regularly.

Shared documentation is critical. Encourage team members to contribute and share knowledge. The entire team should have access to documents. Make it easy for anyone on the team to help keep documents updated.

Document on-call procedures. Make sure on-call duties, schedules, and procedures are documented and shared to all team members. Keep this information up-to-date at all times.

Document escalation procedures for third-party dependencies. If your application depends on external third-party services that you don't directly control, you must have a plan to deal with outages. Create documentation for your planned mitigation processes. Include support contacts and escalation paths.

Use configuration management. Configuration changes should be planned, visible to operations, and recorded. This could take the form of a configuration management database, or a configuration-as-code approach. Configuration should be audited regularly to ensure that what's expected is actually in place.

Get an Azure support plan and understand the process. Azure offers a number of support plans. Determine the right plan for your needs, and make sure the entire team knows how to use it. Team members should understand the details of the plan, how the support process works, and how to open a support ticket with Azure. If you are anticipating a high-scale event, Azure support can assist you with increasing your service limits. For more information, see the Azure Support FAQs.

Follow least-privilege principles when granting access to resources. Carefully manage access to resources. Access should be denied by default, unless a user is explicitly given access to a resource. Only grant a user access to what they need to complete their tasks. Track user permissions and perform regular security audits.

Use Azure role-based access control. Assigning user accounts and access to resources should not be a manual process. Use Azure role-based access control (Azure RBAC) grant access based on Azure Active Directory identities and groups.

Use a bug tracking system to track issues. Without a good way to track issues, it's easy to miss items, duplicate work, or introduce additional problems. Don't rely on informal person-to-person communication to track the status of bugs. Use a bug tracking tool to record details about problems, assign resources to address them, and provide an audit trail of progress and status.

Manage all resources in a change management system. All aspects of your DevOps process should be included in a management and versioning system, so that changes can be easily tracked and audited. This includes code, infrastructure, configuration, documentation, and scripts. Treat all these types of resources as code throughout the test/build/review process.

Use checklists. Create operations checklists to ensure processes are followed. It's common to miss something

in a large manual, and following a checklist can force attention to details that might otherwise be overlooked. Maintain the checklists, and continually look for ways to automate tasks and streamline processes.

For more about DevOps, see What is DevOps? on the Visual Studio site.

Operational Excellence patterns

11/19/2021 • 2 minutes to read • Edit Online

Cloud applications run in a remote datacenter where you do not have full control of the infrastructure or, in some cases, the operating system. This can make management and monitoring more difficult than an on-premises deployment. Applications must expose runtime information that administrators and operators can use to manage and monitor the system, as well as supporting changing business requirements and customization without requiring the application to be stopped or redeployed.

PATTERN	SUMMARY
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
Strangler	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.

Overview of the performance efficiency pillar

11/19/2021 • 2 minutes to read • Edit Online

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. Before the cloud became popular, when it came to planning how a system would handle increases in load, many organizations intentionally provisioned oversized workloads to meet business requirements. This decision made sense in on-premises environments because it ensured *capacity* during peak usage. Capacity reflects resource availability (CPU and memory). Capacity was a major consideration for processes that would be in place for many years.

Just as you need to anticipate increases in load in on-premises environments, you need to expect increases in cloud environments to meet business requirements. One difference is that you may no longer need to make long-term predictions for expected changes to ensure you'll have enough capacity in the future. Another difference is in the approach used to manage performance.

To assess your workload using the tenets found in the Microsoft Azure Well-Architected Framework, reference the Microsoft Azure Well-Architected Review.

To boost performance efficiency, we recommend the following video about optimizing for quick and reliable VM deployments:

Topics

The performance efficiency pillar covers the following topics to help you effectively scale your workload:

PERFORMANCE EFFICIENCY TOPIC	DESCRIPTION
Performance efficiency checklist	Review your application architecture to ensure your workload scales to meet the demands placed on it by users in an efficient manner.
Performance principles	Principles to guide you in your overall strategy for improving performance efficiency.
Design for performance	Review your application architecture from a performance design standpoint.
Consider scalability	Plan for growth by understanding your current workloads.
Plan for capacity	Plan to scale your application tier by adding extra infrastructure to meet demand.
Performance monitoring checklist	Monitor services and check the health state of current workloads to maintain overall workload performance.
Performance patterns	Implement design patterns to build more performant workloads.

PERFORMANCE EFFICIENCY TOPIC	DESCRIPTION
Tradeoffs	Consider tradeoffs between performance optimization and other aspects of the design, such as reliability, security, cost efficiency, and operability.

Next steps

Reference the performance efficiency principles intended to guide you in your overall strategy.

Principles

Performance efficiency principles

11/19/2021 • 3 minutes to read • Edit Online

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. You need to anticipate increases in cloud environments to meet business requirements. These principles are intended to guide you in your overall strategy for improving performance efficiency.

Understand the challenges of distributed architectures

Most cloud deployments are based on distributed architectures where components are distributed across various services. Troubleshooting monolithic applications often requires only one or two lenses—the application and the database. With distributed architectures, troubleshooting is complex and challenging because of various factors. For example, capturing telemetry throughout the application—across all services—as possible. Also, the team should be skilled with the necessary expertise to troubleshoot all services in your architecture.

Run performance testing in the scope of development

Any development effort must go through continuous performance testing. The tests make sure that *any* change to the codebase doesn't negatively affect the application's performance. Establish a regular cadence for running the tests. Run the test as part of a scheduled event or part of a continuous integration (CI) build pipeline.

- **Establish performance baselines**—Determine the current efficiency of the application and its supporting infrastructure. Measuring performance against baselines can provide strategies for improvements and determine if the application is meeting the business goals.
- **Run load and stress tests**—Load testing measures your application's performance under predetermined amounts of load. Stress testing measures the *maximum* load your application and its infrastructure can support before it buckles.
- **Identify bottlenecks**—Bottlenecks is an area within your application that can hinder performance. These spots can be the result of shortages in code or misconfiguration of a service. Typically, a bottleneck worsens as load increases.

Continuously monitor the application and the supporting infrastructure

- **Have a data-driven approach**—Base your decisions on the data captured from repeatable processes. Archive data to monitor performance changes *over time*, not just compared to the last measurement taken.
- **Monitor the health of current workloads**—In monitoring strategy, consider scalability *and* resiliency of the infrastructure, application, and dependent services. For scalability, look at the metrics that would allow you to provision resources dynamically and scale with demand. For reliability, look for early warning signs that might require proactive intervention.
- **Troubleshoot performance issues**—Issues in performance can arise from database queries, connectivity between services, under-provisioned resources, or memory leaks in code. Application telemetry and profiling can be useful tools for troubleshooting your application.

Identify improvement opportunities with resolution planning

Understand the scope of your planned resolution and communicate the changes to all necessary stakeholders. Make code enhancements through a new build. Enhancements to infrastructure may involve many teams. This involved effort may require updated configurations and deprecations in favor of more-appropriate solutions.

Invest in capacity planning

Plan for fluctuation in expected load that can occur because of world events. Test variations of load before the

events, including unexpected ones, to ensure that your application can scale. Make sure all regions can adequately scale to support total load, if a region fails. Take into consideration:

- Technology and the SKU service limits.
- SLAs when determining the services to use in the design. Also, the SLAs of those services.
- Cost analysis to determine how much improvement will be realized in the application if costs are increased. Evaluate if the cost is worth the investment.

Next section

Use this checklist to review your application architecture from a performance design standpoint.

Design checklist

Related links

- Performance efficiency impacts the entire architecture spectrum. Bridge gaps in your knowledge of Azure by reviewing the five pillars of the Microsoft Azure Well-Architected Framework.
- To assess your workload using pillars, see the Microsoft Azure Well-Architected Review.

Checklist - Design for performance efficiency

11/19/2021 • 13 minutes to read • Edit Online

Application design is critical to handling scale as load increases. Design is part of the Performance Efficiency pillar in the Microsoft Azure Well-Architected Framework. Use this checklist to review your application architecture from a performance design standpoint.

Application design

- **Design for scaling**. Scaling allows applications to react to variable load by increasing and decreasing the number of instances of roles, queues, and other services they use. However, the application must be designed with this in mind. During scale operations, application and service instances come and go, and because of this they must be stateless. This prevents the addition or removal of specific instances from adversely affecting current users.
  - You should also implement configuration, autodetection, or load balancing so that as services are added or removed, the application can perform the necessary routing. For example, a web application might use a set of queues in a round-robin approach to route requests to background services running in worker roles. The web application must be able to detect changes in the number of queues, to successfully route requests and balance the load on the application.
- **Scale as a unit**. Plan for additional resources to accommodate growth. For each resource, know the upper scaling limits, and use sharding or decomposition to go beyond these limits. Determine the scale units for the system in terms of well-defined sets of resources. This makes applying scale-out operations easier. It also makes operations less prone to negative impact on the application through limitations imposed by lack of resources in some part of the overall system. For example, adding x number of web and worker roles might require y number of additional queues and z number of storage accounts to handle the additional workload generated by the roles. So a scale unit could consist of x web and worker roles, *y* queues, and *z* storage accounts. Design the application so that it's easily scaled by adding one or more scale units.
- **Take advantage of platform autoscaling features**. Where the hosting platform supports an autoscaling capability, such as Azure Autoscale, prefer it to custom or third party mechanisms unless the built-in mechanism can't fulfill your requirements. Use scheduled scaling rules where possible to ensure resources are available without a start-up delay, but add reactive autoscaling to the rules where appropriate to cope with unexpected changes in demand. You can use the autoscaling operations in the classic deployment model (the older model) to adjust autoscaling, and to add custom counters to rules. For more information, see Auto-scaling guidance.
- **Partition the workload**. Design parts of the process to be discrete and decomposable. Minimize the size of each part, while following the usual rules for separation of concerns and the single responsibility principle. This allows the component parts to be distributed in a way that maximizes use of each compute unit (such as a role or database server). It also makes it easier to scale the application by adding instances of specific resources. For complex domains, consider adopting a microservices architecture.
- **Avoid client affinity**. Where possible, ensure that the application does not require affinity. When you do this, requests can be routed to any instance, and the number of instances is irrelevant. This also avoids the overhead of storing, retrieving, and maintaining state information for each user.
- **Offload CPU-intensive and I/O-intensive tasks as background tasks**. If a request to a service is expected to take a long time to run or absorb considerable resources, offload the processing for this request to a separate task. Use worker roles or background jobs (depending on the hosting platform) to

execute these tasks. This strategy enables the service to continue receiving requests and remain responsive. For more information, see Background jobs guidance.

- **Distribute the workload for background tasks**. Where there are many background tasks, or the tasks require considerable time or resources, spread the work across multiple compute units (such as worker roles or background jobs). For one possible solution, see the Competing Consumers pattern.

- **Consider moving toward a *shared-nothing* architecture**. A shared-nothing architecture uses independent, self-sufficient nodes that have no single point of contention (such as shared services or storage). In theory, such a system can scale almost indefinitely. While a fully shared-nothing approach is generally not practical for most applications, it may provide opportunities to design for better scalability. For example, avoiding the use of server-side session state, client affinity, and data partitioning are good examples of moving toward a shared-nothing architecture.

Data management

- **Use data partitioning**. Divide the data across multiple databases and database servers, or design the application to use data storage services that can provide this partitioning transparently (examples include Azure SQL Database Elastic Database, and Azure Table storage). This approach can help to maximize performance and allow easier scaling. There are different partitioning techniques, such as horizontal, vertical, and functional. You can use a combination of these to achieve maximum benefit from increased query performance, simpler scalability, more flexible management, better availability, and matching the type of store to the data it will hold.

  > **NOTE**
  >
  > Consider using different types of data store for different types of data. Choose the types based on how well they are optimized for the specific type of data. This may include using table storage, a document database, or a column-family data store, instead of, or as well as, a relational database. For more information, see Data partitioning guidance.

- **Design for eventual consistency**. Eventual consistency improves scalability by reducing or removing the time needed to synchronize related data partitioned across multiple stores. The cost is that data is not always consistent when it is read, and some write operations may cause conflicts. Eventual consistency is ideal for situations where the same data is read frequently but written infrequently.

- **Reduce chatty interactions between components and services**. Avoid designing interactions in which an application is required to make multiple calls to a service (each of which returns a small amount of data), rather than a single call that can return all of the data. Where possible, combine several related operations into a single request when the call is to a service or component that has noticeable latency. This makes it easier to monitor performance and optimize complex operations. For example, use stored procedures in databases to encapsulate complex logic, and reduce the number of round trips and resource locking.

- **Use queues to level the load for high velocity data writes**. Surges in demand for a service can overwhelm that service and cause escalating failures. To prevent this, consider implementing the Queue-Based Load Leveling pattern. Use a queue that acts as a buffer between a task and a service that it invokes. This can smooth intermittent heavy loads that may otherwise cause the service to fail or the task to time out.

- **Minimize the load on the data store**. The data store is commonly a processing bottleneck, a costly resource, and often not easy to scale out. Where possible, remove logic (such as processing XML documents or JSON objects) from the data store, and perform processing within the application. For example, instead of passing XML to the database (other than as an opaque string for storage), serialize or

deserialize the XML within the application layer and pass it in a form that is native to the data store.

> **NOTE**
> Typically, it's much easier to scale out the application than the data store, so you should attempt to do as much of the compute-intensive processing as possible within the application.

- **Minimize the volume of data retrieved**. Retrieve only the data you require by specifying columns and using criteria to select rows. Make use of table value parameters and the appropriate isolation level. Use mechanisms such as entity tags to avoid retrieving data unnecessarily.
- **Aggressively use caching**. Use caching wherever possible to reduce the load on resources and services that generate or deliver data. Caching is typically suited to data that is relatively static, or that requires considerable processing to obtain. Caching should occur at all levels where appropriate in each layer of the application, including data access and user interface generation. For more information, see Caching Guidance.
- **Handle data growth and retention**. The amount of data stored by an application grows over time. This growth increases storage costs as well as latency when accessing the data, affecting application throughput and performance. It may be possible to periodically archive some of the old data that is no longer accessed, or move data that is rarely accessed into long-term storage that is more cost efficient, even if the access latency is higher.
- **Optimize Data Transfer Objects (DTOs) using an efficient binary format**. DTOs are passed between the layers of an application many times. Minimizing the size reduces the load on resources and the network. However, balance the savings with the overhead of converting the data to the required format in each location where it is used. Adopt a format that has the maximum interoperability to enable easy reuse of a component.
- **Set cache control**. Design and configure the application to use output caching or fragment caching where possible, to minimize processing load.
- **Enable client side caching**. Web applications should enable cache settings on the content that can be cached. This is commonly disabled by default. Configure the server to deliver the appropriate cache control headers to enable caching of content on proxy servers and clients.
- **Use Azure blob storage and the Azure Content Delivery Network to reduce the load on the application**. Consider storing static or relatively static public content, such as images, resources, scripts, and style sheets, in blob storage. This approach relieves the application of the load caused by dynamically generating this content for each request. Additionally, consider using the Content Delivery Network to cache this content and deliver it to clients. Using the Content Delivery Network can improve performance at the client because the content is delivered from the geographically closest datacenter that contains a Content Delivery Network cache. For more information, see Best practices for using content delivery networks.
- **Optimize and tune SQL queries and indexes**. Some T-SQL statements or constructs may have an adverse effect on performance that can be reduced by optimizing the code in a stored procedure. For example, avoid converting **datetime** types to a **varchar** before comparing with a **datetime** literal value. Use date/time comparison functions instead. Lack of appropriate indexes can also slow query execution. If you use an object/relational mapping framework, understand how it works and how it may affect performance of the data access layer. For more information, see Query Tuning.
- **Consider denormalizing data**. Data normalization helps to avoid duplication and inconsistency. However, maintaining multiple indexes, checking for referential integrity, performing multiple accesses to small chunks of data, and joining tables to reassemble the data imposes an overhead that can affect

performance. Consider if some additional storage volume and duplication is acceptable in order to reduce the load on the data store. Also consider if the application itself (which is typically easier to scale) can be relied on to take over tasks such as managing referential integrity in order to reduce the load on the data store. For more information, see Horizontal, vertical, and functional data partitioning.

Implementation

- **Review the performance antipatterns**. See Performance antipatterns for cloud applications for common practices that are likely to cause scalability problems when an application is under pressure.
- **Use asynchronous calls**. Use asynchronous code wherever possible when accessing resources or services that may be limited by I/O or network bandwidth, or that have a noticeable latency, in order to avoid locking the calling thread.
- **Avoid locking resources, and use an optimistic approach instead**. Never lock access to resources such as storage or other services that have noticeable latency, because this is a primary cause of poor performance. Always use optimistic approaches to managing concurrent operations, such as writing to storage. Use features of the storage layer to manage conflicts. In distributed applications, data may be only eventually consistent.
- **Compress highly compressible data over high latency, low bandwidth networks**. In the majority of cases in a web application, the largest volume of data generated by the application and passed over the network is HTTP responses to client requests. HTTP compression can reduce this considerably, especially for static content. This can reduce cost as well as reduce the load on the network, though compressing dynamic content does apply a fractionally higher load on the server. In more generalized environments, data compression can reduce the volume of data transmitted and minimize transfer time and costs, but the compression and decompression processes incur overhead.

  > **NOTE**
  > Compression should only be used when there is a demonstrable gain in performance. Other serialization methods, such as JSON or binary encodings, may reduce the payload size while having less impact on performance, whereas XML is likely to increase it.

- **Minimize the time that connections and resources are in use**. Maintain connections and resources only for as long as you need to use them. For example, open connections as late as possible, and allow them to be returned to the connection pool as soon as possible. Acquire resources as late as possible, and dispose of them as soon as possible.
- **Minimize the number of connections required**. Service connections absorb resources. Limit the number that are required and ensure that existing connections are reused whenever possible. For example, after performing authentication, use impersonation where appropriate to run code as a specific identity. This can help to make best use of the connection pool by reusing connections.

  > **NOTE**
  > APIs for some services automatically reuse connections, provided service-specific guidelines are followed. It's important that you understand the conditions that enable connection reuse for each service that your application uses.

- **Send requests in batches to optimize network use**. For example, send and read messages in batches when accessing a queue, and perform multiple reads or writes as a batch when accessing storage or a cache. This can help to maximize efficiency of the services and data stores by reducing the number of calls across the network.

- **Avoid a requirement to store server-side session state** where possible. Server-side session state management typically requires client affinity (that is, routing each request to the same server instance), which affects the ability of the system to scale. Ideally, you should design clients to be stateless with respect to the servers that they use. However, if the application must maintain session state, store sensitive data or large volumes of per-client data in a distributed server-side cache that all instances of the application can access.
- **Optimize table storage schemas**. When using table stores that require the table and column names to be passed and processed with every query, such as Azure table storage, consider using shorter names to reduce this overhead. However, do not sacrifice readability or manageability by using overly compact names.
- **Create resource dependencies during deployment or at application startup**. Avoid repeated calls to methods that test the existence of a resource and then create the resource if it does not exist. Methods such as *CloudTable.CreateIfNotExists* and *CloudQueue.CreateIfNotExists* in the Azure Storage Client Library follow this pattern. These methods can impose considerable overhead if they are invoked before each access to a storage table or storage queue. Instead:
  - Create the required resources when the application is deployed, or when it first starts. (A single call to *CreateIfNotExists* for each resource in the startup code for a web or worker role is acceptable.) However, be sure to handle exceptions that may arise if your code attempts to access a resource that doesn't exist. In these situations, you should log the exception, and possibly alert an operator that a resource is missing.
  - Under some circumstances, it may be appropriate to create the missing resource as part of the exception handling code. Adopt this approach with caution, as the non-existence of the resource might indicate a programming error (for example, a misspelled resource name), or some other infrastructure-level issue.
- **Use lightweight frameworks**. Carefully choose the APIs and frameworks you use to minimize resource usage, execution time, and overall load on the application. For example, using Web API to handle service requests can reduce the application footprint and increase execution speed, but it may not be suitable for advanced scenarios where the additional capabilities of Windows Communication Foundation are required.
- **Consider minimizing the number of service accounts**. For example, use a specific account to access resources or services that impose a limit on connections, or perform better where fewer connections are maintained. This approach is common for services such as databases, but it can affect the ability to accurately audit operations due to the impersonation of the original user.

Next steps

Challenges of monitoring distributed architectures

Challenges of monitoring distributed architectures

11/19/2021 • 2 minutes to read • Edit Online

Most cloud deployments are based on distributed architectures where components are distributed across various services. Troubleshooting monolithic applications often requires only one or two lenses—the application and the database. With distributed architectures, troubleshooting is complex and challenging because of various factors. This article describes some of those challenges.

Key points

- The team may not have the expertise across all the services used in an architecture.
- Uncovering and resolving bottlenecks by monitoring all of your services and their infrastructure is complex.
- Antipatterns in design and code causes issues the application is under pressure.
- Resilience any service may impact your application's ability to meet current load.

Team expertise

Distributed architectures require many areas of expertise. To adequately monitor performance, it's critical that telemetry is captured throughout the application, across all services, and is rich. Also, your team should have the necessary skills to troubleshoot all services used in the architecture. When making technology choices, it's advantageous and simple to choose a service over another because of the team's expertise. As the collective skillset grows, you can incorporate other technologies.

Scaling issues

For monolithic applications, scale is two-dimensional. An application usually consist of a group of application servers, some web front ends (WFEs), and database servers. *Uncovering* bottlenecks is simpler but *resolving* them can require considerable effort. For distributed applications, complexity increases exponentially in both aspects for performance issues. You must consider each application, its supporting service, and the latency between all the application layers.

Performance efficiency is a complex mixture of applications and infrastructure (IaaS and PaaS). First, you must ensure that all services can scale to support the expected load and that one service will not cause a bottleneck. Second, while performance testing, you may realize that certain services scale under different load conditions as opposed to scaling all services uniformly. Monitoring all of your services and their infrastructure can help fine-tune your application for optimal performance.

For more information about monitoring for scalability, see Monitor performance for scalability and reliability.

Antipatterns in design

Antipatterns in design and code are a common cause for performance problems when an application is under pressure. For example, an application behaves as expected during performance testing. However, when it's released to production and starts to handle live workloads, performance decreases. Problems such as rejecting user requests, stalling, or throwing exceptions may arise. To learn how to identify and fix these antipatterns, see Performance antipatterns for cloud applications.

Fault tracking

If a service in the architecture fails, how will it affect your application's overall performance? Is the error

transient, allowing your application to continue to function; or, will the application experience a critical failure? If the error is transient, does your application experience a decrease in performance? Resiliency plays a significant role in performance efficiency because the failure of any service may impact your application's ability to meet your business goals and scale to meet current load. Chaos testing—the introduction of random failures within your infrastructure—against your application can help determine how well your application continues to perform under varying stages of load.

For more information about reliability impacts on performance, see Monitor performance for scalability and reliability.

Next

Design scalable Azure applications

Community links

To learn more about chaos testing, see Advancing resilience through chaos engineering and fault injection.

Related links

- Performance antipatterns for cloud applications
- Monitor performance for scalability and reliability

Back to the main article

Design scalable Azure applications

11/19/2021 • 8 minutes to read • Edit Online

Application design is critical to handling scale as load increases. This article will give you insight on the most important topics. For more topics related to handling scale, see the Design Azure applications for efficiency article in the Performance efficiency pillar.

Choose the right data storage

The overall design of the data tier can greatly affect an application's performance and scalability. The Azure storage platform is designed to be massively scalable to meet the data storage and performance needs of modern applications.

Data services in the Azure storage platform are:

- Azure Blob - A massively scalable object store for text and binary data. Includes support for big data analytics through Data Lake Storage Gen2.
- Azure Files - Managed file shares for cloud or on-premises deployments.
- Azure Queue - A messaging store for reliable messaging between application components.
- Azure Tables - A NoSQL store for schemaless storage of structured data.
- Azure Disks - Block-level storage volumes for Azure VMs.

Most cloud workloads adopt the *polyglot* persistence approach. Instead of using one data store service, a mix of technologies is used. Your application will most likely require more than one type of data store depending on your requirements. For examples of when to use these data storage types, see Example scenarios.

Each service is accessed through a storage account. To get started, see Create a storage account.

Database considerations

The choice of database can affect an application's performance and scalability. Database reads and writes involve a network call and storage I/O, both of which are expensive operations. Choosing the right database service to store and retrieve data is therefore a critical decision and must be considered to ensure application scalability. Azure has many database services that will fit most needs. In addition, there are third-party options that can be considered from Azure Marketplace.

To help you choose a database type, determine if the application's storage requirements fit a relational design (SQL) versus a key-value/document/graph design (NoSQL). Some applications may have both a SQL and a NoSQL database for different storage needs. Use the Azure data store decision tree to help you find the appropriate managed data storage solution.

Why use a relational database?

Use a relational database when strong consistency guarantees are important — where all changes are atomic, and transactions always leave the data in a consistent state. However, a relational database generally can't scale out horizontally without sharding the data in some way. Implementing manual sharding can be a time consuming task. Also, the data in relational database must be normalized, which isn't appropriate for every data set.

If a relational database is considered optimal, Azure offers several PaaS options that fully manage hosting and operations of the database. Azure SQL Database can host single databases or multiple databases (Azure SQL Database Managed Instance). The suite of offerings spans requirements that cross performance, scale, size,

resiliency, disaster recovery, and migration compatibility. Azure offers the following PaaS relational database services:

- Azure SQL Database
- Azure Database for MySQL
- Azure Database for PostgreSQL
- Azure Database for MariaDB

Why use a NoSQL database?

Use a NoSQL database when application performance and availability are more important than strong consistency. NoSQL databases are ideal for handling large, unrelated, indeterminate, or rapidly changing data. NoSQL databases have trade-offs. For specifics, see Some challenges with NoSQL databases.

Azure provides two managed services that optimize for NoSQL solutions: Azure Cosmos DB and Azure Cache for Redis. For document and graph databases, Cosmos DB offers extreme scale and performance.

For a detailed description of NoSQL and relational databases, see Understanding the differences.

Choose the right VM size

Choosing the wrong VM size can result in capacity issues as VMs approach their limits. It can also lead to unnecessary cost. To choose the right VM size, consider your workloads, number of CPUs, RAM capacity, disk size, and speed according to business requirements. For a snapshot of Azure VM sizes and their purpose, see Sizes for virtual machines in Azure.

Azure offers the following categories of VM sizes, each designed to run different workloads. Click a category for details.

- General-purpose - Provide balanced CPU-to-memory ratio. Ideal for testing and development, small to medium databases, and low to medium traffic web servers.
- Memory optimized - Offer a high memory-to-CPU ratio that is great for relational database servers, medium to large caches, and in-memory analytics.
- Compute optimized - Have a high CPU-to-memory ratio. These sizes are good for medium traffic web servers, network appliances, batch processes, and application servers.
- GPU optimized - Available with single, multiple, or fractional GPUs. These sizes are designed for compute-intensive, graphics-intensive, and visualization workloads.
- High performance compute - Designed to deliver leadership-class performance, scalability, and cost efficiency for a variety of real-world HPC workloads.
- Storage optimized - Offer high disk throughput and IO, and are ideal for Big Data, SQL, NoSQL databases, data warehousing, and large transactional databases. Examples include Cassandra, MongoDB, Cloudera, and Redis.

You can change the sizing requirements according to your needs and requirements.

Build with microservices

Microservices are a popular architectural style for building applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability. Breaking up larger entities into small discrete pieces alone doesn't ensure sizing and scaling capabilities. Application logic needs to be written to control this.

One of the many benefits of microservices is that they can be scaled independently. This lets you scale out subsystems that require more resources, without scaling out the entire application. Another benefit is fault isolation. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as

any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).

To learn more about the benefits of microservices, see Benefits.

Building with microservices comes with challenges such as development and testing. Writing a small service that relies on other dependent services requires a different approach than writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.

See Challenges for a list of possible drawbacks of a microservice architecture.

Use dynamic service discovery for microservices applications

When there are many separate services or instances of services in play, they will need to receive instructions on who to contact and/or other configuration information. Hard coding this information is flawed, and this is where service discovery steps in. A service instance can spin up and dynamically discover the configuration information it needs to become functional without having that information hard coded.

When combined with an orchestration platform designed to execute and manage microservices such as Kubernetes or Service Fabric, individual services can be right sized, scaled up, scaled down, and dynamically configured to match user demand. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources. Both of these platforms provide built-in services for executing, scaling, and operating a microservices architecture; and one of those key services is discovery and finding where a particular service is running.

Kubernetes supports pod autoscaling and cluster autoscaling. To learn more, see Autoscaling. A Service Fabric architecture takes a different approach to scaling for stateless and stateful services. To learn more, see Scaling considerations.

> **TIP**
> When appropriate, decomposing an application into microservices is a level of decoupling that is an architectural best practice. A microservices architecture can also bring some challenges. The design patterns in Design patterns for microservices can help mitigate these challenges.

Establish connection pooling

Establishing connections to databases is typically an expensive operation that involves establishing an authenticated network connection to the remote database server. This is especially true for applications that open new connections frequently. Use connection pooling to reduce connection latency by reusing existing connections and enable higher database throughput (transactions per second) on the server. By doing this, you avoid the expense of opening a new connection for each request.

Pool size limits

Azure limits the number of network connections a virtual machine or AppService instance can make. Exceeding this limit would cause connections to be slowed down or terminated. With connection pooling, a fixed set of connections are established at the startup time and maintained. In many cases, a default pool size might only consist of a small handful of connections that performs quickly in basic test scenarios, but become a bottleneck under scale when the pool is exhausted. Establishing a pool size that maps to the number of concurrent transactions supported on each application instance is a best practice.

Each database and application platform will have slightly different requirements for the right way to set up and leverage the pool. See SQL Server Connection Pooling for a .NET code example using SQL Server and Azure

Database. In all cases, testing is paramount to ensure a connection pool is properly established and working as designed under load.

> **TIP**
> Use a pool size that uses the same number of concurrent connections. Choose a size that can handle more than the existing connections so you can quickly handle a new request coming in.

Integrated security

Integrated security is a singular unified solution to protect every service that a business runs through a set of common policies and configuration settings. In addition to reducing issues associated with scaling, provisioning, and managing (including higher costs and complexity), integrated security also increases control and overall security. However, there may be times when you may not want to use connection pooling for security reasons. For example, although connection pooling improves the performance of subsequent database requests for a single user, that user cannot take advantage of connections made by other users. It also results in at least one connection per user to the database server.

Measure your business' security requirements against the advantages and disadvantages of connection pooling. To learn more, see Pool fragmentation

Next steps

Application efficiency

Design Azure applications for efficiency

11/19/2021 • 4 minutes to read • Edit Online

Making choices that effect performance efficiency is critical to application design. For additional related topics, see the Design scalable Azure applications article in the Performance efficiency pillar.

Reduce response time with asynchronous programming

The time for the caller to receive a response could range from milliseconds to minutes. During that time, the thread is held by the process until the response comes back, or if an exception happens. This is inefficient because it means that no other requests can be processed during the time waiting for a response. An example when multiple requests in flight is inefficient is a bank account. In this situation, only one resource can operate on the request at the same time. Another example is when connection pools can't be shared, and then all of the requests need separate connections to complete.

Asynchronous programming is an alternative approach. It enables a remote service to be executed without waiting and blocking resources on the client. This is a critical pattern for enabling cloud scalable software and is available in most modern programming languages and platforms.

There are many ways to inject asynchronous programming into an application design. For APIs and services that work across the internet, consider using the Asynchronous Request-Reply pattern. When writing code, remote calls can be asynchronously executed using built-in language constructs like `async` / `await` in .NET C#. Review a language construct example. .NET has other built-in platform support for asynchronous programming with task and event based asynchronous patterns.

Process faster by queuing and batching requests

Similar to asynchronous programming, queuing services has long been used as a scalable mechanism to hand off processing work to a service. Highly scalable queuing services are natively supported in Azure. The queue is a storage buffer located between the caller and the processing service. It takes requests, stores them in a buffer, and queues the requests to provide services around the reliable delivery and management of the queued data.

Using a queue is often the best way to hand off work to a processor service. The processor service receives work by listening on a queue and dequeuing messages. If items to be processed enter too quickly, the queuing service will keep them in the queue until the processing service has available resources and asks for a new work item (message). By leveraging the dynamic nature of Azure Functions, the processor service can easily autoscale on demand as the queue builds up to meet the intake pressure. Developing processor logic with Azure Functions to run task logic from a queue is a common, scalable, and cost effective way to using queuing between a client and a processor.

Azure provides some native first-party queueing services with Azure Storage Queues (simple queuing service based on Azure Storage) and Azure Service Bus (message broker service supporting transactions and reduced latency). Many other third-party options are also available through Azure Marketplace.

To learn more about queue-based Load Leveling, see Queue-based Load Leveling pattern. To compare and contrast queues, see Storage queues and Service Bus queues - compared and contrasted.

Optimize with data compression

A well-known optimization best practice for scaling is to use a compression strategy to compress and bundle web pages or API responses. The idea is to shrink the data returned from a page or API back to the browser or

client app. Compressing the data returned to clients optimizes network traffic and accelerates the application. Azure Front Door can perform file compression, and .NET has built-in framework support for this technique with GZip compression. For more information, see Response compression in ASP.NET Core.

Improve scalability with session affinity

If an application is stateful, meaning that data or state will be stored locally in the instance of the application, it may increase the performance of your application, if you enable session affinity. When session affinity is enabled, subsequent requests to the application will be directed to the same server that processed the first request. If session affinity is not enabled, subsequent requests would be directed to the next available server depending on the load balancing rules. Session affinity allows the instance to have some persistent or cached data/context, which can speed subsequent requests. However, if your application does not store large amounts of state or cached data in memory, session affinity might decrease your throughput because one host could get overloaded with requests, while others are dormant.

> **TIP**
>
> Migrate an Azure Cloud Services application to Azure Service Fabric describes **best practices** about stateless services for an application that is migrated from old Azure Cloud Services to Azure Service Fabric.

Run background jobs to meet integration needs

Many types of applications require background tasks that run independently of the user interface (UI). Examples include batch jobs, intensive processing tasks, and long-running processes such as workflows. Background jobs can be executed without requiring user interaction. The application can start the job and then continue to process interactive requests from users. To learn more, see Background jobs.

Background tasks must offer sufficient performance to ensure they do not block the application, or cause inconsistencies due to delayed operation when the system is under load. Typically, performance is improved by scaling the compute instances that host the background tasks. For a list of considerations, see Scaling and performance considerations.

Logic Apps is a serverless consumption (pay-per-use) service that enables a vast set of out-of-the-box ready-to-use connectors and a long-running workflow engine to enable cloud-native integration needs quickly. Logic Apps is flexible enough for scenarios like running tasks/jobs, advanced scheduling, and triggering. Logic Apps also has advanced hosting options to allow it to run within enterprise restricted cloud environments. Logic Apps can be combined with all other Azure services to complement one another, or it can be used independently.

Like all serverless services, Logic Apps doesn't require VM instances to be purchased, enabled, and scaled up and down. Instead, Logic Apps scale automatically on serverless PaaS provided instances, and a consumer only pays based on usage.

Next steps

Design for scaling

Design for scaling

11/19/2021 • 7 minutes to read • Edit Online

Scalability is the ability of a system to handle increased load. Services covered by Azure Autoscale can scale automatically to match demand to accommodate workload. These services scale out to ensure capacity during workload peaks and return to normal automatically when the peak drops.

To achieve performance efficiency, consider how your application design scales and implement PaaS offerings that have built-in scaling operations.

Two main ways an application can scale include *vertical scaling* and *horizontal scaling*. Vertical scaling (scaling *up*) increases the capacity of a resource, for example, by using a larger virtual machine (VM) size. Horizontal scaling (scaling *out*) adds new instances of a resource, such as VMs or database replicas.

Horizontal scaling has significant advantages over vertical scaling, such as:

- *True cloud scale*: Applications are designed to run on hundreds or even thousands of nodes, reaching scales that aren't possible on a single node.
- *Horizontal scale is elastic*: You can add more instances if load increases, or remove instances during quieter periods.
- Scaling out can be triggered automatically, either on a schedule or in response to changes in load.
- Scaling out may be cheaper than scaling up. Running several small VMs can cost less than a single large VM.
- Horizontal scaling can also improve resiliency, by adding redundancy. If an instance goes down, the application keeps running.

An advantage of vertical scaling is that you can do it without making any changes to the application. But at some point, you'll hit a limit, where you can't scale up anymore. At that point, any further scaling must be horizontal.

Horizontal scale must be designed into the system. For example, you can scale out VMs by placing them behind a load balancer. But each VM in the pool must handle any client request, so the application must be stateless or store state externally (say, in a distributed cache). Managed PaaS services often have horizontal scaling and autoscaling built in. The ease of scaling these services is a major advantage of using PaaS services.

Just adding more instances doesn't mean an application will scale, however. It might push the bottleneck somewhere else. For example, if you scale a web front end to handle more client requests, that might trigger lock contentions in the database. You'd want to consider other measures, such as optimistic concurrency or data partitioning, to enable more throughput to the database.

Always conduct performance and load testing to find these potential bottlenecks. The stateful parts of a system, such as databases, are the most common cause of bottlenecks, and require careful design to scale horizontally. Resolving one bottleneck may reveal other bottlenecks elsewhere.

Use the Performance efficiency checklist to review your design from a scalability standpoint.

In the cloud, the ability to take advantage of scalability depends on your infrastructure and services. Platforms, such as Kubernetes, were built with scaling in mind. Virtual machines may not scale as easily although scale operations are possible. With virtual machines, you may want to plan ahead to avoid scaling infrastructure in the future to meet demand. Another option is to select a different platform such as Azure virtual machines scale sets.

When using scalability, you can predict the current average and peak times for your workload. Payment plan options allow you to manage this prediction. You pay either per minute or per-hour depending on the service for a chosen time period.

Plan for growth

Planning for growth starts with understanding your current workloads, which can help you anticipate scale needs based on predictive usage scenarios. An example of a predictive usage scenario is an e-commerce site that recognizes that its infrastructure should scale appropriately for an expected high volume of holiday traffic.

Perform load tests and stress tests to determine the necessary infrastructure to support the predicted spikes in workloads. A good plan includes incorporating a buffer to accommodate for random spikes.

For more information on how to determine the upper and maximum limits of an application's capacity, reference Performance testing in the performance efficiency pillar.

Another critical component of planning for scale is to make sure the region that hosts your application supports the necessary capacity required to accommodate load increase. If you're using a multiregion architecture, make sure the secondary regions can also support the increase. A region can offer the product, but may not support the predicted load increase without the necessary SKUs (Stock Keeping Units) so you need to verify capacity.

To verify your region and available SKUs, first select the product and regions in Products available by region.

TABLE KEY: ✓ Generally Available | In Preview | In Preview (hover to view expected timeframe) | Future availability (hover to view expected timeframe)

	EUROPE		UNITED STATES							
Products	North Europe	West Europe	Central US	East US	East US 2	North Central US	South Central US	West Central US	West US	West US 2
Azure Data Share	✓	✓	◷	✓	✓	◷	✓		◷	✓
Snapshot Execution	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Release dates, features and requirements are subject to change prior to final commercial release of the products/features/software described herein. This page is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESSED, IMPLIED, OR STATUTORY, AS TO THE INFORMATION ON THIS PAGE. To see which regions Microsoft Azure Offers may be eligible, see Find Azure credit offers in your region.

Then, check the SKUs available in the Azure portal.

Add scale units

For each resource, know the upper scaling limits, and use sharding or decomposition to go beyond those limits. Design the application so that it's easily scaled by adding one or more scale units, such as by using the Deployment Stamps pattern. Determine the scale units for the system for well-defined sets of resources.

The next step might be to use built-in scaling features or tools to understand which resources need to scale concurrently with other resources. For example, adding X number of front-end VMs might require Y number of extra queues and Z number of storage accounts to handle the extra workload. So a scale unit could consist of X VM instances, Y queues, and Z storage accounts.

Use Autoscaling to manage load increases and decreases

Autoscaling enables you to run the right amount of resources to handle the load of your app. It adds resources (called scaling out) to handle an increase in load such as seasonal workloads. Autoscaling saves money by removing idle resources (called scaling in) during a decrease in load such as nights and weekends for some corporate apps.

You automatically scale between the minimum and maximum number of instances to run, and add, or remove VMs automatically based on a set of rules.

For more information, see Autoscaling.

Understand scale targets

Scale operations (horizontal - changing the number of identical instances, vertical - switching to more/less powerful instances) can be fast, but usually take time to complete. It's important to understand how this delay affects the application under load and if degraded performance is acceptable.

For more information, reference Best practices for Autoscale.

Take advantage of platform autoscaling features

Here's how you can benefit from autoscaling features:

- Use built-in autoscaling features when possible rather than custom or third-party mechanisms.
- Use scheduled scaling rules where possible to ensure that resources are available.
- Add reactive autoscaling to the rules where appropriate to cope with unexpected changes in demand.

> **NOTE**
> If your application is explicitly designed to handle the termination of some of its instances, ensure you use autoscaling to scale down and scale in resources no longer necessary for the given load to reduce operational costs.

For more information, reference Autoscaling.

Autoscale CPU or memory-intensive applications

CPU or memory-intensive applications require scaling up to a larger machine SKU with more CPU or memory. Once you've reduced the demand for CPU or memory, instances can revert back to the original instance.

For example, you may have an application that processes images, videos, or music. Given the process and requirements, it may make sense to scale up a server by adding CPU or memory to quickly process the large media file. While scaling *out* allows the system to process more files simultaneously, it won't impact processing speed of each individual file.

Autoscale with Azure compute services

Autoscaling works by collecting metrics for the resource (CPU and memory usage), and the application (requests queued and requests per second). Rules can then be created to add and remove instances depending on how the rule evaluates. An App Services app plan allows autoscale rules to be set for scale-out/scale-in and scale-up/scale-down. Scaling also applies to Azure Automation.

The Application Service autoscaling sample shows how to create an Azure App Service plan, which includes an Azure App Service.

Azure Kubernetes Service (AKS) offers two levels of autoscale:

- **Horizontal autoscale**: Can be enabled on service containers to add more or fewer pod instances within the cluster.
- **Cluster autoscale**: Can be enabled on the agent VM instances running an agent node-pool to add more or remove VM instances dynamically.

Other Azure services include the following services:

- **Azure Service Fabric**: Virtual machine scale sets offer autoscale capabilities for true IaaS scenarios.
- **Azure App Gateway** and **Azure API Management**: PaaS offerings for ingress services that enable

autoscale.

- **Azure Functions**, **Azure Logic Apps**, and **App Services**: Serverless pay-per-use consumption modeling that inherently provides autoscaling capabilities.
- **Azure SQL Database**: PaaS platform to change performance characteristics of a database on the fly and assign more resources when needed or release the resources when they aren't needed. Allows scaling up/down, read scale-out, and global scale-out/sharding capabilities.

Each service documents its autoscale capabilities. Review Autoscale overview for a general discussion on Azure platform autoscale.

> **NOTE**
>
> If your application doesn't have built-in ability to autoscale, or isn't configured to scale out automatically as load increases, it's possible that your application's services will fail if they become saturated with user requests. Reference Azure Automation for possible solutions.

Next steps

Plan for capacity

Plan for capacity

11/19/2021 • 4 minutes to read • Edit Online

Azure offers many options to meet capacity requirements as your business grows. These options can also minimize cost.

Scale out rather than scaling up

When using cloud technologies, it's generally easier, cheaper, and more effective to scale out than scaling up. Plan to scale your application tier by adding extra infrastructure to meet demand. Be sure to remove the resources when they are not needed. If you plan to scale up by increasing the resources allocated to your hosts, you will reach a limit where it becomes cost-prohibitive to scale any further. Scaling up also often requires downtime for your servers to reboot.

Prepare infrastructure for large-scale events

Large-scale application design takes careful planning and possibly involves complex implementation. Work with your business and marketing teams to prepare for large-scale events. Knowing if there will be sudden spikes in traffic such as Superbowl, Black Friday, or Marketing pushes, can allow you to prepare your infrastructure ahead of time.

A fundamental design principle in Azure is to scale out by adding machines or service instances based on increased demand. Scaling out can be a better alternative to purchasing additional hardware, which may not be in your budget. Depending on your payment plan, you don't pay for idle VMs or need to reserve capacity in advance. A pay-as-you-go plan is usually ideal for applications that need to meet planned spikes in traffic.

> **NOTE**
>
> Don't plan for capacity to meet the highest level of expected demand. An inappropriate or misconfigured service can impact cost. For example, building a multiregion service when the service levels don't require high-availability or geo-redundancy will increase cost without reasonable business justification.

Choose the right resources

Right sizing your infrastructure to meet the needs of your applications can save you considerably as opposed to a "one size fits all" solution often employed with on-premises hardware. You can choose various options when you deploy Azure VMs to support workloads.

Each VM type has specific features and different combinations of CPU, memory, and disks. For example, the B-series VMs are ideal for workloads that don't need the full performance of the CPU continuously, like web servers, proof of concepts, small databases, and development build environments. The B-Series offers a cost effective way to deploy these workloads that don't need the full performance of the CPU continuously and burst in their performance.

For a list of sizes and a description of the recommended use, see sizes for virtual machines in Azure.

Continually monitor workloads after migration to find out if your VMs aren't optimized or have frequent periods when they aren't used. If you discover this, it makes sense to either shut down the VMs or downscale them by using virtual machine scale sets. You can optimize a VM with Azure Automation, virtual machine scale sets, auto-shutdown, and scripted or third-party solutions. To learn more, see Automate VM optimization.

Along with choosing the right VMs, selecting the right storage type can save your organization significant cost every month. For a list of storage data types, access tiers, storage account types, and storage redundancy options, see Select the right storage.

Use metrics to fine-tune scaling

It's often difficult to understand the relationship between metrics and capacity requirements, especially when an application is initially deployed. Provision a little extra capacity at the beginning, and then monitor and tune the *autoscaling* rules to bring the capacity closer to the actual load. *Autoscaling* enables you to run the right amount of resources to handle the load of your app. It adds resources (called scaling out) to handle an increase in load such as seasonal workloads and customer facing applications.

After configuring the autoscaling rules, monitor the performance of your application over time. Use the results of this monitoring to adjust the way in which the system scales if necessary.

Azure Monitor autoscale provides a common set of autoscaling functionality for virtual machine scale sets, Azure App Service, and Azure Cloud Service. Scaling can be performed on a schedule, or based on a runtime metric, such as CPU or memory usage. For example, you can scale out by one instance if average CPU usage is above 70%, and scale in by one instance if CPU usage falls below 50 percent.

The default autoscaling rules are set to know when it's time to execute an autoscaling action in order to prevent the system from reacting too quickly. To learn more, see Autoscaling.

For a list of built-in metrics, see Azure Monitor autoscaling common metrics. You can also implement custom metrics by using Application Insights to monitor the performance of your live applications. Some Azure services use different scaling methods.

Preemptively scaling based on trends

Preemptively scaling based on historical data can ensure your application has consistent performance, even though your metrics haven't yet indicated the need to scale. Schedule-based rules allow you to scale when you see time patterns in your load and want to scale before a possible load increase or decrease occurs. For example, you can set a trigger attribute to scale out to 10 instances on weekdays, and scale in to four (4) instances on Saturday and Sunday. If you can predict the load on the application, consider using scheduled autoscaling, which adds and removes instances to meet anticipated peaks in demand.

To learn more, see Use Azure Monitor autoscale.

Next steps

Performance testing

Capacity planning: When performance testing, the business must communicate any fluctuation in expected load. Load can be impacted by world events, such as political, economic, or weather changes; by marketing initiatives, such as sales or promotions; or, by seasonal events, such as holidays. You should test variations of load prior to events, including unexpected ones, to ensure that your application can scale. Additionally, you should ensure that all regions can adequately scale to support total load, should one region fail.

Checklist - Testing for performance efficiency

11/19/2021 • 5 minutes to read • Edit Online

Is the application tested for performance, scalability, and resiliency?

Performance testing helps to maintain systems properly and fix defects before problems reach system users. It's part of the Performance Efficiency pillar in the Microsoft Azure Well-Architected Framework.

Performance testing is the superset of both load and stress testing. The primary goal of performance testing is to validate benchmark behavior for the application.

Load Testing validates application scalability by rapidly or gradually increasing the load on the application until it reaches a threshold.

Stress Testing is a type of negative testing, which involves various activities to overload existing resources and remove components. This testing enables you to understand overall resiliency and how the application responds to issues.

Use the following checklist to review your application architecture from a performance testing standpoint.

Performance testing

- **Ensure solid performance testing with shared *team* responsibility**. Successfully implementing meaningful performance tests requires a number of resources. It's not just a single developer or QA Analyst running some tests on their local machine. Instead, performance tests need a test environment (also known as a *test bed*) that tests can be executed against without interfering with production environments and data. Performance testing requires input and commitment from developers, architects, database administrators, and network administrators.

- **Capacity planning**. When performance testing, the business must communicate any fluctuation in expected load. Load can be impacted by world events, such as political, economic, or weather changes; by marketing initiatives, such as sales or promotions; or, by seasonal events, such as holidays. Test variations of load prior to events, including unexpected ones, to ensure that your application can scale. Additionally, you should ensure that all regions can adequately scale to support total load, should one region fail.

- **Identify a path forward to leveraging existing tests or the creation of new tests**. There are different types of performance testing: load testing, stress testing, API testing, client-side/browser testing, and so on. It's important that you understand and articulate the different types of tests, along with their advantages and disadvantages, to the customer.

- **Perform testing in all stages in the development and deployment life cycle**. Application code, infrastructure automation, and fault tolerance should all be tested. This can ensure that the application will perform as expected in every situation. You'll want to test early enough in the application life cycle to catch and fix errors. Errors are cheaper to repair when caught early and can be expensive or impossible to fix later. To learn more, reference Testing your application and Azure environment.

- **Avoid experiencing poor performance with testing**. Two subsets of performance testing, load testing and stress testing, can determine the upper limit, and maximum point of failure, respectively, of the application's capacity. By performing these tests, you can determine the necessary infrastructure to support the anticipated workloads.

- **Plan for a load buffer to accommodate random spikes without overloading the infrastructure**. For example, if a normal system load is `100,000` requests per second, the infrastructure

should support `100,000` requests at `80%` of total capacity (for example, `125,000` requests per second). If you expect that the application will continue to sustain `100,000` requests per second, and the current SKU (Stock Keeping Unit) introduces latency at `65,000` requests per second, you'll most likely need to upgrade your product to the next higher SKU. If there is a secondary region, you'll need to ensure that it also supports the higher SKU.

- **Test failover in multiregions**. Test the amount of time it would take for users to be rerouted to the paired region so that the region doesn't fail. Typically, a planned test failover can help determine how much time would be required to fully scale to support the redirected load.
- **Configure the environment based on testing results to sustain performance efficiency**. Scale out or scale in to handle increases and decreases in load. For example, you may know that you'll encounter high levels of traffic during the day and low levels on weekends. You may configure the environment to scale out for increases in load or scale in for decreases before the load actually changes.

Testing tools

- **Choose testing tools based on the type of performance testing you are attempting to execute**. There are various performance testing tools available for DevOps. Some tools like `JMeter` only perform testing against endpoints and tests HTTP statuses. Other tools such as K6 and Selenium can perform tests that also check data quality and variations. Application Insights, while not necessarily designed to test server load, can test the performance of an application within the user's browser.
- **Carry out performance profiling and load testing** during development, as part of test routines, and before final release to ensure the application performs and scales as required. This testing should occur on the same type of hardware as the production platform, and with the same types, and quantities of data, and user load as it will encounter in production.
- **Determine if it is better to use automated or manual testing**. Testing can be automated or manual. Automating tests is the best way to make sure that they're executed. Depending on how frequently tests are performed, they're typically limited in duration and scope. Manual testing is run much less frequently.
- **Cache data to improve performance, scalability, and availability**. The more data that you have, the greater the benefits of caching become. Caching typically works well with data that is immutable or that changes infrequently.
- **Decide how you will handle local development and testing when some static content is expected to be served from a content delivery network (CDN)**. For example, you could pre-deploy the content to the CDN as part of your build script. Instead, use compile directives or flags to control how the application loads the resources. For example, in debug mode, the application could load static resources from a local folder. In release mode, the application would use the CDN.
- **Simulate different workloads on your application and measure application performance for each workload**. This technique is the best way to figure out what resources you will need to host your application. Use performance indicators to assess whether your application is performing as expected or not.

Recommendation

Define a testing strategy. For more information, reference Testing.

Next steps

Performance testing

Performance testing

11/19/2021 • 5 minutes to read • Edit Online

Performance testing helps to maintain systems properly and fix defects before problems reach system users. It helps maintain the efficiency, responsiveness, scalability, and speed of applications when compared with business requirements. When done effectively, performance testing should give you the diagnostic information necessary to eliminate bottlenecks, which lead to poor performance. A bottleneck occurs when data flow is either interrupted or stops due to insufficient capacity to handle the workload.

To avoid experiencing poor performance, commit time and resources to testing system performance. Two subsets of performance testing, load testing and stress testing, can determine the upper (close to capacity limit) and maximum (point of failure) limit, respectively, of the application's capacity. By performing these tests, you can determine the necessary infrastructure to support the anticipated workloads.

A best practice is to plan for a load buffer to accommodate random spikes without overloading the infrastructure. For example, if a normal system load is 100,000 requests per second, the infrastructure should support 100,000 requests at 80% of total capacity (i.e., 125,000 requests per second). If you anticipate that the application will continue to sustain 100,000 requests per second, and the current SKU (Stock Keeping Unit) introduces latency at 65,000 requests per second, you'll most likely need to upgrade your product to the next higher SKU. If there is a secondary region, you'll need to ensure that it also supports the higher SKU.

Establish baselines

First, establish performance baselines for your application. Then, establish a regular cadence for running the tests. Run the test as part of a scheduled event or part of a continuous integration (CI) build pipeline.

Baselines help to determine the current efficiency state of your application and its supporting infrastructure. Baselines can provide good insights for improvements and determine if the application is meeting business goals. Baselines can be created for any application regardless of its maturity. No matter when you establish the baseline, measure performance against that baseline during continued development. When code and, or infrastructure changes, the effect on performance can be actively measured.

Load testing

Load testing measures system performance as the workload increases. It identifies where and when your application breaks, so you can fix the issue before shipping to production. It does this by testing system behavior under typical and heavy loads.

Load testing takes places in stages of load. These stages are usually measured by virtual users (VUs) or simulated requests, and the stages happen over given intervals. Load testing provides insights into how and when your application needs to scale in order to continue to meet your SLA to your customers (whether internal or external). Load testing can also be useful for determining latency across distributed applications and microservices.

The following are key points to consider for load testing:

- **Know the Azure service limits:** Different Azure services have *soft* and *hard* limits associated with them. The terms soft limit and hard limit describe the current, adjustable service limit (soft limit) and the maximum limit (hard limit). Understand the limits for the services you consume so that you are not blocked if you need to exceed them. For a list of the most common Azure limits, see Azure subscription and service limits, quotas, and constraints.

The ResourceLimits sample shows how to query the limits and quotas for commonly used resources.

- **Measure typical loads:** Knowing the typical and maximum loads on your system helps you understand when something is operating outside of its designed limits. Monitor traffic to understand application behavior.
- **Understand application behavior under various scales:** Load test your application to understand how it performs at various scales. First, test to see how the application performs under a typical load. Then, test to see how it performs under load using different scaling operations. To get additional insight into how to evaluate your application as the amount of traffic sent to it increases, see Autoscale best practices.

Stress testing

Unlike load testing, which ensures that a system can handle what it's designed to handle, stress testing focuses on overloading the system until it breaks. A stress test determines how stable a system is and its ability to withstand extreme increases in load. It does this by testing the maximum number requests from another service (for example) that a system can handle at a given time before performance is compromised and fails. Find this maximum to understand what kind of load the current environment can adequately support.

Determine the maximum demand you want to place on memory, CPU, and disk IOPS. Once a stress test has been performed, you will know the maximum supported load and an operational margin. It is best to choose an operational threshold so that scaling can be performed before the threshold has been reached.

Once you determine an acceptable operational margin and response time under typical loads, verify that the environment is configured adequately. To do this, make sure the SKUs that you selected are based on the desired margins. Be careful to stay as close as possible to your margins. Allocating too much can increase costs and maintenance unnecessarily; allocating too few can result in poor user experience.

In addition to stress testing through increased load, you can stress test by reducing resources to identify what happens when the machine runs out of memory. You can also stress test by increasing latency (e.g., the database takes 10x time to reply, writes to storage takes 10x longer, etc.).

Multiregion testing

A multiregion architecture can provide higher availability than deploying to a single region. If a regional outage affects the primary region, you can use Front Door to use the secondary region. This architecture can also help if an individual subsystem of the application fails.

Test the amount of time it would take for users to be rerouted to the paired region so that the region doesn't fail. To learn more about routing, see Front Door routing methods. Typically, a planned test failover can help determine how much time would be required to fully scale to support the redirected load.

Configure the environment based on testing results

Once you have performed testing and found an acceptable operational margin and response under increased levels of load, configure the environment to sustain performance efficiency. Scale out or scale in to handle increases and decreases in load. For example, you may know that you will encounter high levels of traffic during the day and low levels on weekends. You may configure the environment to scale out for increases in load or scale in for decreases before the load actually changes.

For more information on autoscaling, see Design for scaling in the Performance Efficiency pillar.

> **NOTE**
> Ensure that a rule has been configured to scale the environment back down once load reaches below the set thresholds. This will save you money.

Next steps

Testing tools

Testing tools

11/19/2021 • 4 minutes to read • Edit Online

There are multiple stages in the development and deployment life cycle in which tests can be performed. Application code, infrastructure automation, and fault tolerance should all be tested. This can ensure that the application will perform as expected in every situation. You'll want to test early enough in the application life cycle to catch and fix errors. Errors are cheaper to repair when caught early and can be expensive or impossible to fix later.

Testing can be automated or manual. Automating tests is the best way to make sure that they are executed. Depending on how frequently tests are performed, they are typically limited in duration and scope. Manual testing is run much less frequently. For a list of tests that you should consider while developing and deploying applications, see Testing your application and Azure environment.

Identify baselines and goals for performance

Knowing where you are (baseline) and where you want to be (goal) makes it easier to plan how to get there. Established baselines and goals will help you to stay on track and measure progress. Testing may also uncover a need to perform additional testing on areas that you may not have planned.

Baselines can vary based on connections or platforms used to accessing the application. It may be important to establish baselines that address the different connections, platforms, and elements such as time of day, or weekday versus weekend.

There are many types of goals when determining baselines for application performance. Some examples are, the time it takes to render a page, or a desired number of transactions if your site conducts e-commerce. The following list shows some examples of questions that may help you to determine goals.

What are your baselines and goals for:

- Establishing an initial connection to a service?
- An API endpoint complete response?
- Server response times?
- Latency between systems/microservices?
- Database queries?

Caching data

Caching can dramatically improve performance, scalability, and availability. The more data that you have, the greater the benefits of caching become. Caching typically works well with data that is immutable or that changes infrequently. Examples include reference information such as product and pricing information in an e-commerce application, or shared static resources that are costly to construct. Some or all of this data can be loaded into the cache at application startup to minimize demand on resources and to improve performance.

Use performance testing and usage analysis to determine whether pre-populating or on-demand loading of the cache, or a combination of both, is appropriate. The decision should be based on the volatility and usage pattern of the data. Cache utilization and performance analysis are particularly important in applications that encounter heavy loads and must be highly scalable.

To learn more about how to use caching as a solution in testing, see Caching.

Use Azure Redis to cache data

Azure Cache for Redis is a caching service that can be accessed from any Azure application, whether the application is implemented as a cloud service, a website, or inside an Azure virtual machine. Caches can be shared by client applications that have the appropriate access key. It is a high-performance caching solution that provides availability, scalability, and security.

To learn more about using Azure Cache for Redis, see Considerations for implementing caching in Azure.

Content delivery network

Content delivery networks (CDNs) are typically used to deliver static content such as images, style sheets, documents, client-side scripts, and HTML pages. The major advantages of using a CDN are lower latency and faster delivery of content to users, regardless of their geographical location in relation to the datacenter where the application is hosted. CDNs can help to reduce load on a web application because the application does not have to service requests for the content that is hosted in the CDN. Using a CDN is a good way to minimize the load on your application, and maximize availability and performance. Consider adopting this strategy for all of the appropriate content and resources your application uses.

Decide how you will handle local development and testing when some static content is expected to be served from a CDN. For example, you could pre-deploy the content to the CDN as part of your build script. Alternatively, use compile directives or flags to control how the application loads the resources. For example, in debug mode, the application could load static resources from a local folder. In release mode, the application would use the CDN.

To learn more about CDNs, see Best practices for using content delivery networks (CDNs).

Benchmark testing

Benchmarking is the process of simulating different workloads on your application and measuring application performance for each workload. It is the best way to figure out what resources you will need to host your application. Use performance indicators to assess whether your application is performing as expected or not. Take into consideration VM sizes and disk sizes.

See the Optimize IOPS, throughput, and latency table for guidance.

Metrics

Metrics measure trends over time. They are available for interactive analysis in the Azure portal with Azure Metrics Explorer. Metrics also can be added to an Azure dashboard for visualization in combination with other data and used for near-real time alerting.

Performance testing gives you the ability to see specific details on the processing capabilities of applications. You'll most likely want a monitoring tool that allows you to discover proactively if the issues you find through testing are appearing in both your infrastructure and applications. Azure Monitor Metrics is a feature of Azure Monitor that collects metrics from monitored resources into a time series database.

With Azure Monitor, you can collect, analyze, and act on telemetry from your cloud and on-premises environments. It helps you understand how applications are performing and identifies issues affecting them and the resources they depend on.

For a list of Azure metrics, see Supported metrics with Azure Monitor.

Next steps

Performance monitoring

Monitoring for performance efficiency

11/19/2021 • 2 minutes to read • Edit Online

Troubleshooting an application's performance requires monitoring and reliable investigation. Issues in performance can arise from database queries, connectivity between services, under-provisioned resources, or memory leaks in code.

Continuously monitoring services and checking the health state of current workloads is key in maintaining the overall performance of the workload. An overall monitoring strategy consider these factors:

- Scalability
- Resiliency of the infrastructure, application, and dependent services
- Application and infrastructure performance

Checklist

How are you monitoring to ensure the workload is scaling appropriately?

- Enable and capture telemetry throughout your application to build and visualize end-to-end transaction flows for the application.
- See metrics from Azure services such as CPU and memory utilization, bandwidth information, current storage utilization, and more.
- Use resource and platform logs to get information about what events occur and under which conditions.
- For scalability, look at the metrics to determine how to provision resources dynamically and scale with demand.
- In the collected logs and metrics look for signs that might make a system or its components suddenly become unavailable.
- Use log aggregation technology to gather information across all application components.
- Store logs and key metrics of critical components for statistical evaluation and predicting trends.
- Identify antipatterns in the code.

In this section

Follow these questions to assess the workload at a deeper level.

ASSESSMENT	DESCRIPTION
Are application logs and events correlated across all application components?	Correlate logs and and events for subsequent interpretation. This will give you visibility into end-to-end transaction flows.
Are you collecting Azure Activity Logs within the log aggregation tool?	Collect platform metrics and logs to get visibility into the health and performance of services that are part of the architecture.
Are application and resource level logs aggregated in a single data sink, or is it possible to cross-query events at both levels?)	Implement a unified solution to aggregate and query application and resource level logs, such as Azure Log Analytics.

Azure services

The monitoring operations should utilize Azure Monitor. You can analyze data, set up alerts, get end-to-end views of your applications, and use machine learning–driven insights to identify and resolve problems quickly. Export logs and metrics to services such as Azure Log Analytics or an external service like Splunk. Furthermore, application technologies such as Application Insights can enhance the telemetry coming out of applications.

Next section

Based on insights gained through monitoring, optimize your code. One option might be to consider other Azure services that may be more appropriate for your objectives.

Optimize

Related links

Back to the main article

Performance Efficiency patterns

11/19/2021 • 2 minutes to read • Edit Online

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. You need to anticipate these increases to meet business requirements. An important consideration in achieving performance efficiency is to consider how your application scales and to implement PaaS offerings that have built-in scaling operations. Scalability is ability of a system either to handle increases in load without impact on performance or for the available resources to be readily increased. It concerns not just compute instances, but other elements such as data storage, messaging infrastructure, and more.

PATTERN	SUMMARY
Cache-Aside	Load data on demand into a cache from a data store.
Choreography	Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control.
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads.
Sharding	Divide a data store into a set of horizontal partitions or shards.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.

PATTERN	SUMMARY
Throttling	Control the consumption of resources used by an instance of an application, an individual tenant, or an entire service.

Performance efficiency checklist

11/19/2021 • 14 minutes to read • Edit Online

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner, and is one of the pillars of the Microsoft Azure Well-Architected Framework. Use this checklist to review your application architecture from a performance efficiency standpoint.

Application design

Partition the workload. Design parts of the process to be discrete and decomposable. Minimize the size of each part, while following the usual rules for separation of concerns and the single responsibility principle. This allows the component parts to be distributed in a way that maximizes use of each compute unit (such as a role or database server). It also makes it easier to scale the application by adding instances of specific resources. For complex domains, consider adopting a microservices architecture.

Design for scaling. Scaling allows applications to react to variable load by increasing and decreasing the number of instances of roles, queues, and other services they use. However, the application must be designed with this in mind. For example, the application and the services it uses must be stateless, to allow requests to be routed to any instance. This also prevents the addition or removal of specific instances from adversely affecting current users. You should also implement configuration or autodetection of instances as they are added and removed, so that code in the application can perform the necessary routing. For example, a web application might use a set of queues in a round-robin approach to route requests to background services running in worker roles. The web application must be able to detect changes in the number of queues, to successfully route requests and balance the load on the application.

Scale as a unit. Plan for additional resources to accommodate growth. For each resource, know the upper scaling limits, and use sharding or decomposition to go beyond these limits. Determine the scale units for the system in terms of well-defined sets of resources. This makes applying scale-out operations easier, and less prone to negative impact on the application through limitations imposed by lack of resources in some part of the overall system. For example, adding x number of web and worker roles might require y number of additional queues and z number of storage accounts to handle the additional workload generated by the roles. So a scale unit could consist of x web and worker roles, *y* queues, and *z* storage accounts. Design the application so that it's easily scaled by adding one or more scale units. Consider using the Deployment Stamps pattern to deploy scale units.

Avoid client affinity. Where possible, ensure that the application does not require affinity. Requests can thus be routed to any instance, and the number of instances is irrelevant. This also avoids the overhead of storing, retrieving, and maintaining state information for each user.

Take advantage of platform autoscaling features. Where the hosting platform supports an autoscaling capability, such as Azure Autoscale, prefer it to custom or third-party mechanisms unless the built-in mechanism can't fulfill your requirements. Use scheduled scaling rules where possible to ensure resources are available without a start-up delay, but add reactive autoscaling to the rules where appropriate to cope with unexpected changes in demand. You can use the autoscaling operations in the classic deployment model to adjust autoscaling, and to add custom counters to rules. For more information, see Auto-scaling guidance.

Offload CPU-intensive and I/O-intensive tasks as background tasks. If a request to a service is expected to take a long time to run or absorb considerable resources, offload the processing for this request to a separate task. Use worker roles or background jobs (depending on the hosting platform) to execute these tasks. This strategy enables the service to continue receiving further requests and remain responsive. For more information, see Background jobs guidance.

Distribute the workload for background tasks. Where there are many background tasks, or the tasks require considerable time or resources, spread the work across multiple compute units (such as worker roles or background jobs). For one possible solution, see the Competing Consumers pattern.

Consider moving toward a *shared-nothing* architecture. A shared-nothing architecture uses independent, self-sufficient nodes that have no single point of contention (such as shared services or storage). In theory, such a system can scale almost indefinitely. While a fully shared-nothing approach is generally not practical for most applications, it may provide opportunities to design for better scalability. For example, avoiding the use of server-side session state, client affinity, and data partitioning are good examples of moving toward a shared-nothing architecture.

Data management

Use data partitioning. Divide the data across multiple databases and database servers, or design the application to use data storage services that can provide this partitioning transparently (examples include Azure SQL Database Elastic Database, and Azure Table storage). This approach can help to maximize performance and allow easier scaling. There are different partitioning techniques, such as horizontal, vertical, and functional. You can use a combination of these to achieve maximum benefit from increased query performance, simpler scalability, more flexible management, better availability, and to match the type of store to the data it will hold. Also, consider using different types of data store for different types of data, choosing the types based on how well they are optimized for the specific type of data. This may include using table storage, a document database, or a column-family data store, instead of, or as well as, a relational database. For more information, see Data partitioning guidance.

Design for eventual consistency. Eventual consistency improves scalability by reducing or removing the time needed to synchronize related data partitioned across multiple stores. The cost is that data is not always consistent when it is read, and some write operations may cause conflicts. Eventual consistency is ideal for situations where the same data is read frequently but written infrequently. For more information, see the Data Consistency Primer.

Reduce chatty interactions between components and services. Avoid designing interactions in which an application is required to make multiple calls to a service (each of which returns a small amount of data), rather than a single call that can return all of the data. Where possible, combine several related operations into a single request when the call is to a service or component that has noticeable latency. This makes it easier to monitor performance and optimize complex operations. For example, use stored procedures in databases to encapsulate complex logic, and reduce the number of round trips and resource locking.

Use queues to level the load for high velocity data writes. Surges in demand for a service can overwhelm that service and cause escalating failures. To prevent this, consider implementing the Queue-Based Load Leveling pattern. Use a queue that acts as a buffer between a task and a service that it invokes. This can smooth intermittent heavy loads that may otherwise cause the service to fail or the task to time out.

Minimize the load on the data store. The data store is commonly a processing bottleneck, a costly resource, and often not easy to scale out. Where possible, remove logic (such as processing XML documents or JSON objects) from the data store, and perform processing within the application. For example, instead of passing XML to the database (other than as an opaque string for storage), serialize or deserialize the XML within the application layer and pass it in a form that is native to the data store. It's typically much easier to scale out the application than the data store, so you should attempt to do as much of the compute-intensive processing as possible within the application.

Minimize the volume of data retrieved. Retrieve only the data you require by specifying columns and using criteria to select rows. Make use of table value parameters and the appropriate isolation level. Use mechanisms like entity tags to avoid retrieving data unnecessarily.

Aggressively use caching. Use caching wherever possible to reduce the load on resources and services that

generate or deliver data. Caching is typically suited to data that is relatively static, or that requires considerable processing to obtain. Caching should occur at all levels where appropriate in each layer of the application, including data access and user interface generation. For more information, see the Caching Guidance.

Handle data growth and retention. The amount of data stored by an application grows over time. This growth increases storage costs as well as latency when accessing the data, affecting application throughput and performance. It may be possible to periodically archive some of the old data that is no longer accessed, or move data that is rarely accessed into long-term storage that is more cost efficient, even if the access latency is higher.

Optimize Data Transfer Objects (DTOs) using an efficient binary format. DTOs are passed between the layers of an application many times. Minimizing the size reduces the load on resources and the network. However, balance the savings with the overhead of converting the data to the required format in each location where it is used. Adopt a format that has the maximum interoperability to enable easy reuse of a component.

Set cache control. Design and configure the application to use output caching or fragment caching where possible, to minimize processing load.

Enable client side caching. Web applications should enable cache settings on the content that can be cached. This is commonly disabled by default. Configure the server to deliver the appropriate cache control headers to enable caching of content on proxy servers and clients.

Use Azure blob storage and the Azure Content Delivery Network to reduce the load on the application. Consider storing static or relatively static public content, such as images, resources, scripts, and style sheets, in blob storage. This approach relieves the application of the load caused by dynamically generating this content for each request. Additionally, consider using the Content Delivery Network to cache this content and deliver it to clients. Using the Content Delivery Network can improve performance at the client because the content is delivered from the geographically closest datacenter that contains a Content Delivery Network cache. For more information, see the Content Delivery Network Guidance.

Optimize and tune SQL queries and indexes. Some T-SQL statements or constructs may have an adverse effect on performance that can be reduced by optimizing the code in a stored procedure. For example, avoid converting **datetime** types to a **varchar** before comparing with a **datetime** literal value. Use date/time comparison functions instead. Lack of appropriate indexes can also slow query execution. If you use an object/relational mapping framework, understand how it works and how it may affect performance of the data access layer. For more information, see Query Tuning.

Consider denormalizing data. Data normalization helps to avoid duplication and inconsistency. However, maintaining multiple indexes, checking for referential integrity, performing multiple accesses to small chunks of data, and joining tables to reassemble the data imposes an overhead that can affect performance. Consider if some additional storage volume and duplication is acceptable in order to reduce the load on the data store. Also consider if the application itself (which is typically easier to scale) can be relied on to take over tasks such as managing referential integrity in order to reduce the load on the data store. For more information, see Data partitioning guidance.

Implementation

Review the performance antipatterns. See Performance antipatterns for cloud applications for common practices that are likely to cause scalability problems when an application is under pressure.

Use asynchronous calls. Use asynchronous code wherever possible when accessing resources or services that may be limited by I/O or network bandwidth, or that have a noticeable latency, in order to avoid locking the calling thread.

Avoid locking resources, and use an optimistic approach instead. Never lock access to resources such as storage or other services that have noticeable latency, because this is a primary cause of poor performance. Always use optimistic approaches to managing concurrent operations, such as writing to storage. Use features

of the storage layer to manage conflicts. In distributed applications, data may be only eventually consistent.

Compress highly compressible data over high latency, low bandwidth networks. In the majority of cases in a web application, the largest volume of data generated by the application and passed over the network is HTTP responses to client requests. HTTP compression can reduce this considerably, especially for static content. This can reduce cost as well as reducing the load on the network, though compressing dynamic content does apply a fractionally higher load on the server. In other, more generalized environments, data compression can reduce the volume of data transmitted and minimize transfer time and costs, but the compression and decompression processes incur overhead. As such, compression should only be used when there is a demonstrable gain in performance. Other serialization methods, such as JSON or binary encodings, may reduce the payload size while having less impact on performance, whereas XML is likely to increase it.

Minimize the time that connections and resources are in use. Maintain connections and resources only for as long as you need to use them. For example, open connections as late as possible, and allow them to be returned to the connection pool as soon as possible. Acquire resources as late as possible, and dispose of them as soon as possible.

Minimize the number of connections required. Service connections absorb resources. Limit the number that are required and ensure that existing connections are reused whenever possible. For example, after performing authentication, use impersonation where appropriate to run code as a specific identity. This can help to make best use of the connection pool by reusing connections.

> **NOTE**
>
> APIs for some services automatically reuse connections, provided service-specific guidelines are followed. It's important that you understand the conditions that enable connection reuse for each service that your application uses.

Send requests in batches to optimize network use. For example, send and read messages in batches when accessing a queue, and perform multiple reads or writes as a batch when accessing storage or a cache. This can help to maximize efficiency of the services and data stores by reducing the number of calls across the network.

Avoid a requirement to store server-side session state where possible. Server-side session state management typically requires client affinity (that is, routing each request to the same server instance), which affects the ability of the system to scale. Ideally, you should design clients to be stateless with respect to the servers that they use. However, if the application must maintain session state, store sensitive data or large volumes of per-client data in a distributed server-side cache that all instances of the application can access.

Optimize table storage schemas. When using table stores that require the table and column names to be passed and processed with every query, such as Azure table storage, consider using shorter names to reduce this overhead. However, do not sacrifice readability or manageability by using overly compact names.

Create resource dependencies during deployment or at application startup. Avoid repeated calls to methods that test the existence of a resource and then create the resource if it does not exist. Methods such as *CloudTable.CreateIfNotExists* and *CloudQueue.CreateIfNotExists* in the Azure Storage Client Library follow this pattern. These methods can impose considerable overhead if they are invoked before each access to a storage table or storage queue. Instead:

- Create the required resources when the application is deployed, or when it first starts (a single call to *CreateIfNotExists* for each resource in the startup code for a web or worker role is acceptable). However, be sure to handle exceptions that may arise if your code attempts to access a resource that doesn't exist. In these situations, you should log the exception, and possibly alert an operator that a resource is missing.
- Under some circumstances, it may be appropriate to create the missing resource as part of the exception handling code. But you should adopt this approach with caution as the non-existence of the resource might be indicative of a programming error (a misspelled resource name for example), or some other

infrastructure-level issue.

Use lightweight frameworks. Carefully choose the APIs and frameworks you use to minimize resource usage, execution time, and overall load on the application. For example, using Web API to handle service requests can reduce the application footprint and increase execution speed, but it may not be suitable for advanced scenarios where the additional capabilities of Windows Communication Foundation are required.

Consider minimizing the number of service accounts. For example, use a specific account to access resources or services that impose a limit on connections, or perform better where fewer connections are maintained. This approach is common for services such as databases, but it can affect the ability to accurately audit operations due to the impersonation of the original user.

Carry out performance profiling and load testing during development, as part of test routines, and before final release to ensure the application performs and scales as required. This testing should occur on the same type of hardware as the production platform, and with the same types and quantities of data and user load as it will encounter in production. For more information, see Testing the performance of a cloud service.

Tradeoffs for performance efficiency

11/19/2021 • 6 minutes to read • Edit Online

As you design the workload, consider tradeoffs between performance optimization and other aspects of the design, such as cost efficiency, operability, reliability, and security.

Performance efficiency vs. cost efficiency

Cost can increase as a result of boosting performance. Here are a few factors to consider when optimizing for performance and how they impact cost:

- Avoid cost estimation of a workload at consistently high utilization. Consumption-based pricing will be more expensive than the equivalent provisioned pricing. Smooth out the peaks to get a consistent flow of compute and data. Ideally, use manual and autoscaling to find the right balance. Scaling up is generally more expensive than scaling out.
- Cost scales directly with number of regions. Locating resources in cheaper regions shouldn't negate the cost of network ingress and egress or degraded application performance because of increased latency.
- Every render cycle of a payload consumes both compute and memory. You can use caching to reduce load on servers and save with pre-canned storage and bandwidth costs. The savings can be dramatic, especially for static content services.
  - While caching can reduce cost, there are some performance tradeoffs. For example, Azure Traffic Manager pricing is based on the number of DNS (Domain Name Service) queries that reach the service. You can reduce that number through caching and configure how often the cache is refreshed. Relying on the cache that isn't frequently updated will cause longer user failover times if an endpoint is unavailable.
- Using dedicated resources for batch processing long running jobs will increase the cost. You can lower cost by provisioning Spot VMs but be prepared for the job to be interrupted every time Azure evicts the VM.

For cost considerations, see the Cost Optimization pillar.

Performance efficiency vs. operational excellence

As you determine how to scale your workload to meet the demands placed on it by users in an efficient manner, consider the operations processes that are keeping an application running in production. To achieve operational excellence with these processes, make sure the deployments remain reliable and predictable. They should be automated to reduce the chance of human error. They should be a fast and routine process, so they don't slow down the release of new features or bug fixes. Equally important, you must be able to quickly roll back or roll forward if an update has problems.

Automated performance testing

One operational process that can help to identify performance issues early is automated performance testing. The impact of a serious performance issue can be as severe as a bug in the code. While automated functional tests can prevent application bugs, they might not detect performance problems. Define acceptable performance goals for metrics such as latency, load times, and resource usage. Include automated performance tests in your release pipeline, to make sure the application meets those goals.

Fast builds

Another operational efficiency process is making sure that your product is in a deployable state through a fast build process. Builds provide crucial information about the status of your product.

The following can help faster builds:

- Select the right size of VMs.
- Ensure that the build server is located near the sources and a target location, so it can reduce the duration of your build considerably.
- Scale-out build servers.
- Optimizing the build.

For an explanation of these items, see Builds.

Monitoring performance optimization

As you consider making performance improvements, monitoring should be done to verify that your application is running correctly. Monitoring should include the application, platform, and networking. To learn more, see Monitoring.

For operational considerations, see the Operational Excellence pillar.

Performance efficiency vs. reliability

We acknowledge up front that failures will happen. Instead of trying to prevent failures altogether, the goal is to minimize the effects of a single failing component.

Reliable applications are *resilient* and *highly available* (HA). Resiliency allows systems to recover gracefully from failures, and they continue to function with minimal downtime and data loss before full recovery. HA systems run as designed in a healthy state with no significant downtime. Maintaining reliability enables you to maintain performance efficiency.

Some reliability considerations are:

- Use the Circuit Breaker pattern to provide stability while the system recovers from a failure and minimizes the impact on performance.
- Achieve levels of scale and performance needed for your solution by segregating read and write interfaces by implementing the CQRS pattern.
- Often, you can achieve higher availability by adopting an *eventual consistency* model. To learn about selecting the correct data store, see Use the best data store for the job.
- If your application requires more storage accounts than are currently available in your subscription, create a new subscription with additional storage accounts. For more information, see Scalability and performance targets.
- Avoid scaling up or down. Instead, select a tier and instance size that meet your performance requirements under typical load, and then scale out the instances to handle changes in traffic volume. Scaling up and down may trigger an application restart.
- Create a separate storage account for logs. Don't use the same storage account for logs and application data. This helps to prevent logging from reducing application performance.
- Monitor performance. Use a performance monitoring service such as Application Insights or New Relic to monitor application performance and behavior under load. Performance monitoring gives you real-time insight into the application. It enables you to diagnose issues and perform root-cause analysis of failures.

For resiliency, availability, and reliability considerations, see the Reliability pillar.

Performance efficiency vs. security

If performance is so poor that the data is unusable, you can consider the data to be inaccessible. From a security perspective, you need to do whatever you can to make sure that your services have optimal uptime and performance.

A popular and effective method for enhancing availability and performance is load balancing. Load balancing is a method of distributing network traffic across servers that are part of a service. It helps performance because the processor, network, and memory overhead for serving requests are distributed across all the load-balanced servers. We recommend that you employ load balancing whenever you can, and as appropriate for your services. For information on load balancing scenarios, see Optimize uptime and performance.

Consider these security measures, which impact performance:

- To optimize performance and maximize availability, application code should first try to get OAuth access tokens silently from a cache before attempting to acquire a token from the identity provider. OAuth is a technological standard that allows you to securely share information between services without exposing your password.
- Ensure that you are integrating critical security alerts and logs into SIEMs (security information and event management) without introducing a high volume of low value data. Doing so can increase SIEM cost, false positives, and lower performance. For more information, see Prioritize alert and log integration.
- Use Azure AD Connect to synchronize your on-premises directory with your cloud directory. There are factors that affect the performance of Azure AD Connect. Ensure Azure AD Connect has enough capacity to keep underperforming systems from impeding security and productivity. Large or complex organizations (organizations provisioning more than 100,000 objects) should follow the recommendations to optimize their Azure AD Connect implementation.
- If you want to gain access to real time performance information at the packet level, use packet capture to set alerts.

For other security considerations, see the Security pillar.

Overview of a hybrid workload

11/19/2021 • 4 minutes to read • Edit Online

Customer workloads are becoming increasingly complex, with many applications often running on different hardware across on-premises, multicloud, and the edge. Managing these disparate workload architectures, ensuring uncompromised security, and enabling developer agility are critical to success.

Azure uniquely helps you meet these challenges, giving you the flexibility to innovate anywhere in your hybrid environment while operating seamlessly and securely. The Well-Architected Framework includes a hybrid description for each of the five pillars: cost optimization, operational excellence, performance efficiency, reliability, and security. These descriptions create clarity on the considerations needed for your workloads to operate effectively across hybrid environments.

Adopting a hybrid model offers multiple solutions that enable you to confidently deliver hybrid workloads: run Azure data services anywhere, modernize applications anywhere, and manage your workloads anywhere.

Extend Azure management to any infrastructure

> **TIP**
>
> Applying the principles in this article series to each of your workloads will better prepare you for hybrid adoption. For larger or centrally managed organizations, hybrid and multicloud are commonly part of a broader strategic objective. If you need to scale these principle across a portfolio of workloads using hybrid and multicloud environments, you may want to start with the Cloud Adoption Framework's hybrid and multicloud scenario and best practices. Then return to this series to refine each of your workload architectures.

Use *Azure Arc enabled infrastructure* to extend Azure management to any infrastructure in a hybrid environment. Key features of Azure Arc enabled infrastructure are:

- **Unified Operations**
  - Organize resources such as virtual machines, Kubernetes clusters and Azure services deployed across your entire IT environment.
  - Manage and govern resources with a single pane of glass from Azure.
  - Integrate with Azure Lighthouse for managed service provider support.
- **Adopt cloud practices**
  - Easily adopt DevOps techniques such as infrastructure as code.
  - Empower developers with self-service and choice of tools.
  - Standardize change control with configuration management systems, such as GitOps and DSC.

Run Azure services anywhere

Azure Arc allows you to run Azure Services anywhere. This allows you to build consistent hybrid and multicloud application archtiectures by using Azure services that can run in Azure, on-premises, at the edge, or at other cloud providers.

Run Azure data services anywhere

Use *Azure Arc enabled data services* to run Azure data services anywhere to support your hybrid workloads. Key features of Azure Arc enabled data services are:

- Run Azure data services on any Kubernetes cluster deployed on any hardware.
- Gain cloud automation benefits, always up-to-date innovation in Azure data services, unified management of your on-premises and cloud data assets with a cloud billing model across both environments.
- Azure SQL Database and Azure PostgreSQL Hyperscale are the first set of Azure data services that are Azure Arc enabled.

Run Azure Application services anywhere

Use *Azure Arc enabled Application services* to run Azure App Service, Functions, Logic Apps, Event Grid, and API Management anywhere to support your hybrid workloads. Key features of Azure Arc enabled application services are as follows:

- Web Apps - Azure App Service makes building and managing web applications and APIs easy, with a fully managed platform and features like autoscaling, deployment slots, and integrated web authentication.
- Functions - Azure Functions makes event-driven programming simple, with state-of-the-art autoscaling, and with triggers and bindings to integrate with other Azure services.
- Logic Apps - Azure Logic Apps produces automated workflows for integrating apps, data, services, and backend systems, with a library of more than 400 connectors.
- Event Grid - Azure Event Grid simplifies event-based applications, with a single service for managing the routing of events from any source to any destination.
- Azure API Management gateway - Azure API Management provides a unified management experience and full observability across all internal and external APIs.

Modernize applications anywhere

Use the *Azure Stack family* to modernize applications without ever leaving the datacenter. Key features of the Azure Stack family are:

- Extend Azure to your on-premises workloads with Azure Stack Hub. Build and run cloud apps on premises, in connected or disconnected scenarios, to meet regulatory or technical requirements.
- Use Azure Stack HCI to run virtualized workloads on premises and easily connect to Azure to access cloud management and security services.
- Build and run your intelligent edge solutions on Azure Stack Edge, an Azure managed appliance to run machine learning models and compute at the edge to get results quickly—and close to where data is being generated. Easily transfer the full data set to Azure for further analysis or archive.

Manage workloads anywhere

Use *Azure Arc management* to extend Azure management to all assets in your workloads, regardless of where they are hosted. Key features of Azure Arc management are:

- **Adopt cloud practices**
  - Easily adopt DevOps techniques such as infrastructure as code.
  - Empower developers with self-service and choice of tools.
  - Standardize change control with configuration management systems, such as GitOps and DSC.
- **Scale across workloads with Unified Operations**
  - Organize resources such as virtual machines, Kubernetes clusters and Azure services deployed across your entire IT environment.
  - Manage and govern resources with a single pane of glass from Azure.
  - Integrate with Azure Lighthouse for managed service provider support.

Next steps

Cost optimization in a hybrid workload

11/19/2021 • 4 minutes to read • Edit Online

A key benefit of hybrid cloud environments is the ability to scale dynamically and back up resources in the cloud, avoiding the capital expenditures of a secondary datacenter. However, when workloads sit in both on-premises and cloud environments, it can be challenging to have visibility into the cost. With Azure's hybrid technologies, you can define policies and constraints for both on-premises and cloud workloads with Azure Arc. By utilizing Azure Policy, you're able to enforce organizational standards for your workload and the entire IT estate.

Azure Arc helps minimize or even eliminate the need for on-premises management and monitoring systems, which reduces operational complexity and cost, especially in large, diverse, and distributed environments. This helps offset additional costs associated with Azure Arc-related services. For example, advanced data security for Azure Arc enabled SQL Server instance requires Microsoft Defender for Cloud functionality of Microsoft Defender for Cloud, which has pricing implications.

Other considerations are described in the Principles of cost optimization section in the Microsoft Azure Well-Architected Framework.

Workload definitions

Define the following for your workloads:

- **Monitor cloud spend with hybrid workloads**. Track cost trends and forecast future spend with dashboards in Azure for your on-prem data estates with Azure Arc.
- **Keep within cost constraints**.
  - Create, apply, and enforce standardized and custom tags and policies.
  - Enforce run-time conformance and audit resources with Azure Policy.
- **Choose a flexible billing model**. With Azure Arc enabled data services, you can use existing hardware with the addition of an operating expense (OPEX) model.

Functionality

For budget concerns, you get a considerable amount of functionality at no cost that you can use across all of your servers and cluster with Azure Arc enabled servers. You can turn on additional Azure services to each workload as you need them, or not at all.

- **Free Core Azure Arc capabilities**
  - Update, management
  - Search index
  - Group, tags
  - Portal
  - Templates, extensions
  - RBAC, subscriptions
- **Paid-for Azure Arc enabled attached services**
  - Azure policy
  - Azure monitor
  - Defender for Cloud – Standard

- Microsoft Sentinel
- Backup
- Config and change management

Tips

- **Start slow**. Light up new capabilities as needed. Most of Azure Arc's resources are free to start.
- **Save time with unified management** for your on-premises and cloud workloads by projecting them all into Azure.
- **Automate and delegate** remediation of incidents and problems to service teams without IT intervention.

Azure Architecture Center (AAC) resources related to hybrid cost

- Manage configurations for Azure Arc enabled servers
- Azure Arc hybrid management and deployment for Kubernetes clusters
- Optimize administration of SQL Server instances in on-premises and multi-cloud environments by leveraging Azure Arc
- Disaster Recovery for Azure Stack Hub virtual machines
- Build high availability into your BCDR strategy
- Use Azure Stack HCI switchless interconnect and lightweight quorum for Remote Office/Branch Office
- Archive on-premises data to cloud

Infrastructure Decisions

Azure Stack HCI can help in cost-savings by using your existing Hyper-V and Windows Server skills to consolidate aging servers and storage. Azure Stack HCI pricing follows the monthly subscription billing model, with a flat rate per physical processor core in an Azure Stack HCI cluster.

Use Azure Stack HCI to modernize on-prem workloads with hyperconverged infra. Azure Stack HCI billing is based on a monthly subscription fee per physical processor core, not a perpetual license. When customers connect to Azure, the number of cores used is automatically uploaded and assessed for billing purposes. Cost doesn't vary with consumption beyond the physical processor cores. This means that more VMs don't cost more, and customers who are able to run denser virtual environments are rewarded.

If you are currently using VMware, you can take advantage of cost savings only available with Azure VMware Solution. Easily move VMware workloads to Azure and increase your productivity with elasticity, scale, and fast provisioning cycles. This will help enhance your workloads with the full range of Azure compute, monitor, backup, database, IoT, and AI services.

Lastly, you can slowly begin migrating out of your datacenter and use Azure Arc while you're migrating to project everything into Azure.

Capacity planning

Check out our checklist under the Cost Optimization pillar in the Well-Framework to learn more about capacity planning, and build a checklist to design cost-effective workloads.

- Define SLAs
- Determine regulatory needs

Provision

One advantage of cloud computing is the ability to use the PaaS model. And in some cases, PaaS services can be cheaper than managing VMs on your own. Some workloads cannot be moved to the cloud though for regulatory or latency reasons. Therefore, using a service like Azure Arc enabled services allows you to flexibly

use cloud innovation where you need it by deploying Azure services anywhere.

Click the following links for guidance in provisioning:

- Azure Arc pricing
- Azure Arc Jumpstart for templates (in GitHub)
- Azure Stack HCI pricing
  - Azure Stack HCI can reduce costs by saving in server, storage, and network infrastructure.
- Azure VMware Solution pricing - Run your VMware workloads natively on Azure
  - Run your VMware workloads natively on Azure.
- Azure Stack Hub pricing

Monitor and optimize

Treat cost monitoring and optimization as a process, rather than a point-in-time activity. You can conduct regular cost reviews and forecast the capacity needs so that you can provision resources dynamically and scale with demand.

- Managing the Azure Arc enabled servers agent
  - Bring all your resources into a single system so you can organize and inventory through a variety of Azure scopes, such as Management groups, Subscriptions, and Resource Groups.
  - Create, apply, and enforce standardized and custom tags to keep track of resources.
  - Build powerful queries and search your global portfolio with Azure Resource Graph.
- With Azure Stack HCI
  - Costs for datacenter real estate, electricity, personnel, and servers can be reduced or eliminated.
  - Costs are now part of OPEX, which can be scaled as needed.

Next steps

Operational excellence

Operational excellence in a hybrid workload

11/19/2021 • 5 minutes to read • Edit Online

Operational excellence consists of the operations processes that keep a system running in production. Applications must be designed with DevOps principles in mind, and deployments must be reliable and predictable. Use monitoring tools to verify that your application is running correctly and to gather custom business telemetry that will tell you whether your application is being used as intended.

Use *Azure Arc enabled infrastructure* to add support for cloud Operational Excellence practices and tools to any environment. Be sure to utilize reference architectures and other resources from this section that illustrate applying these principles in hybrid and multicloud scenarios. The architectures referenced here can also be found in the Azure Architecture Center, Hybrid and Multicloud category.

Build cloud native apps anywhere, at scale

To keep your systems running, many workload teams have architected and designed applications where components are distributed across public cloud services, private clouds, data centers, and edge locations. With Azure Arc enabled Kubernetes, you can accelerate development by using best in class application services with standardized deployment, configuration, security, and observability. One of the primary benefits of Azure Arc is facilitating implementation of DevOps principles that apply established development practices to operations. This results in improved agility, without jeopardizing the stability of IT environment.

- Centrally code and deploy applications confidently to any Kubernetes distribution in any location.
- Centrally manage and delegate access for DevOps roles and responsibilities.
- Reduce errors with consistent configuration and policy-driven deployment and operations for applications and Kubernetes clusters.
- Delegate access for DevOps roles and responsibilities through Azure RBAC.
- Reduce errors with consistent policy driven deployment and operations through GitHub and Azure Policy.

Connect Kubernetes clusters to Azure and start deploying using a GitOps model

GitOps relies on a Git repository to host files that contain the configuration representing the expected state of a resource. An agent running on the cluster monitors the state of the repository and, when there is a change on the repository, the agent pulls the changed files to the cluster and applies the new configuration.

In the context of Azure Arc enabled Kubernetes clusters, a Git repository hosts a configuration of a Kubernetes cluster, including its resources such as pods and deployments. A pod or a set of pods running on the cluster polls the status of the repository and, once it detects a change, it pulls and applies the new configuration to the cluster.

Azure Arc enabled Kubernetes clusters rely on Flux, an open-source GitOps deployment tool to implement the pods responsible for tracking changes to the Git repository you designate and applying them to the local cluster. In addition, the containerized Flux operator also periodically reviews the existing cluster configuration to ensure that it matches the one residing in the Git repository. If there is a configuration drift, the Flux agent remediates it by reapplying the desired configuration.

Each association between an Azure Arc enabled Kubernetes cluster configuration and the corresponding GitOps repository resides in Azure, as part of the Azure Resource Manager resource representing the Azure Arc enabled Kubernetes clusters. You can configure that association via traditional Azure management interfaces, such as the

Azure portal or Azure CLI. Alternatively, you can use Azure Policy to automate this process, allowing you to apply it consistently to all resources in an entire subscription or individual resource groups you designate.

Modernize applications anywhere with Azure Kubernetes Service on Azure Stack HCI

If you are looking for a fully managed Kubernetes solution on-premises in your datacenters and/or edge locations, AKS on Azure Stack HCI is a great option. Azure Kubernetes Service on Azure Stack HCI is an on-premises implementation of Azure Kubernetes Service (AKS), which automates running containerized applications at scale. Azure Kubernetes Service is now in preview on Azure Stack HCI and Windows Server 2019 Datacenter, making it quicker to get started hosting Linux and Windows containers in your datacenter.

AKS clusters on Azure Stack HCI can be connected to Azure Arc for centralized management. Once connected, you can deploy your applications and Azure data services to these clusters and extend Azure services such as Azure Monitor, Azure Policy and Microsoft Defender for Cloud.

Azure Stack HCI use cases

- **Modernize your high-performance workloads and containerized applications**
  - Use Azure Stack HCI to enable automated deployment, scaling and management of containerized applications by running a Kubernetes cluster on your hyperconverged infrastructure.
  - Deploy AKS on Azure Stack HCI using Windows Admin Center or PowerShell.
- **Deploy and manage workloads in remote and branch sites**
  - Use Azure Stack HCI to deploy your container-built edge workloads, and essential business applications in highly available virtual machines (VMs).
  - Bring efficient application development and deployment to remote locations at the right price by leveraging switchless deployment and 2 node clusters.
  - Get a global view of your system's health using Azure Monitor.
- **Upgrade your infrastructure for remote work using VDI**
  - Bring desktops on-premises for low latency and data sovereignty enabling remote work using a brokerage service like Microsoft Remote Desktop Services. With Azure Stack HCI you can scale your resources in a simple predictable way. Provide a secure way to deliver desktop services to a wide range of devices without allowing users to store data locally or upload data from those local devices.

Resources and architectures related to Operational Excellence

The introduction of cloud computing had a significant impact on how software is developed, delivered, and run. With *Azure Arc enabled infrastructure* and Azure Arc components like Azure Arc enabled Kubernetes and Azure Arc enabled data services it becomes possible to design cloud native applications with a consistent set of principles and tooling across public cloud, private cloud, and the edge.

Click the following links for architecture details and diagrams that enable application design and DevOps practices consistent with Operational excellence principles.

Application design

- Azure Arc hybrid management and deployment for Kubernetes clusters
- Run containers in a hybrid environment
- Managing K8 clusters outside of Azure with Azure Arc
- Optimize administration of SQL Server instances in on-premises and multi-cloud environments by leveraging Azure Arc
- Azure Data Studio dashboards

- microsoft/azure_arc: Azure Arc environments bootstrapping for everyone (in github.com)
- All Azure Architecture Center Hybrid and Multicloud Architectures

Monitoring

- Enable monitoring of Azure Arc enabled Kubernetes cluster
- Azure Monitor for containers overview

Application performance management

- Hybrid availability and performance monitoring

Manage data anywhere

Built-in Capabilities	Deployment Model		
	Customer Infrastructure		Azure
	SQL Server / PostgreSQL	Azure Arc enabled databases	Azure PaaS databases
Database security features	✓	✓	✓
Elastic / "Limitless" scalability	✗	✓ Limited by the capacity of customer infrastructure	✓
Automatic HA/DR	✗	✓ Customer responsible for underlying HW/K8s availability	✓
Auto upgrade, patching	✗	✓	✓
Auto backup-restore	✗	✓	✓
Monitoring	✗	✓	✓
Compliance certifications	✗ Customer responsible for compliance certification	✗ Customer responsible for compliance certification	✓ 90+ certifications
Data sovereignty	✓	✓	✓ Azure regions not available in all countries yet
Customer control	✓	✓	✓ Pre-defined HW options No control over HW/OS
Fully managed by Microsoft	✗ Customer-managed	✗ Customer-managed using software provided by Microsoft	✓
Guaranteed availability SLA	✗ Customer-managed	✗ Customer-managed using software provided by Microsoft	✓

Next steps

Performance Efficiency

Performance efficiency in a hybrid workload

11/19/2021 • 5 minutes to read • Edit Online

Performance efficiency is the ability of your workload to scale to meet the demands placed on it by users in an efficient manner. In a hybrid environment, it is important to consider how you manage your on-premises or multicloud workloads to ensure they can meet the demands for scale. You have options to scale up into the cloud when your on-premises resources reach capacity. Scale up, down, and scale out your databases without application downtime.

Using a tool like Azure Arc, you can build cloud native apps anywhere, at scale. Architect and design hybrid applications where components are distributed across public cloud services, private clouds, data centers and edge locations without sacrificing central visibility and control. Deploy and configure applications and Kubernetes clusters consistently and at scale from source control and templates. You can also bring PaaS services on premises. This allows you to use cloud innovation flexibly, where you need it by deploying Azure services anywhere. Implement cloud practices and automation to deploy faster, consistently, and at scale with always up-to-date Azure Arc enabled services. You can scale elastically based on capacity, with the ability to deploy in seconds.

Azure Arc design

The first steps with Azure Arc are to connect the machines to Azure. To use Azure Arc to connect the machine to Azure, you need to install the Azure Connected Machine agent on each machine that you plan to connect using Azure Arc. You can connect any other physical or virtual machine running Windows or Linux to Azure Arc.

There are four ways to connect machines:

1. Manual installation
2. Script-based installation
3. Connect machines at scale using service principal
4. Installation using Windows PowerShell DSC

After connecting the machines, you can then manage the VM extensions all from Azure, which provides consistent extension management between Azure and non-Azure VMs. In Azure you can use Azure Automation State Configuration to centrally store configurations and maintain the desired state of Arc enabled servers through the DSC VM extension. You can also collect log data for analysis with Azure Monitor Logs enabled through the Log Analytics agent VM extension. With Azure Monitor, you can analyze the performance of your Windows and Linux VMs and monitor their processes and dependencies on other resources and external processes.

Azure Arc enabled Kubernetes

With Azure Arc enabled Kubernetes, you need to register the cluster first. You can register any CNCF Kubernetes cluster that is running. You'll need a kubeconfig file to access the cluster and cluster-admin role on the cluster for deploying Arc-enabled Kubernetes agents. You'll use Azure Command-Line Interface (Azure CLI) to perform cluster registration tasks.

Azure Arc enabled SQL Managed Instance

When planning for deployment of Azure Arc enabled SQL Managed Instance, you should identify the correct amount of compute, memory, and storage that will be required to run the Azure Arc data controller and the

intended SQL managed instance server groups.

You have the flexibility to extend the capacity of the underlying Kubernetes or AKS cluster over time by adding additional compute nodes or storage. Kubernetes or AKS offers an abstraction layer over the underlying virtualization stack and hardware. Storage classes implement such abstraction for storage.

> **NOTE**
> When provisioning a pod, you need to decide which storage class to use for its volumes. Your decision is important from a performance standpoint because an incorrect choice could result in suboptimal performance.

When planning for deployment of Azure Arc enabled SQL Managed Instance, you should consider a range of factors affecting storage configuration kubernetes-storage-class-factors for both data controller and database instances.

Azure Stack HCI

With the scope of Azure Arc extended to Azure Stack HCI VMs, you'll be able to automate their configuration by using Azure VM extensions and evaluate their compliance with industry regulations and corporate standards by using Azure Policy.

In remote office/branch office scenarios, you must consider storage resiliency versus usage efficiency, versus performance. Planning for Azure Stack HCI volumes involves identifying the optimal balance between resiliency, usage efficiency, and performance. The challenge results from the fact that maximizing one of these characteristics typically has a negative impact on at least one of the other two.

To learn more, see Use Azure Stack HCI switchless interconnect and lightweight quorum for Remote Office/Branch Office.

Monitoring in a hybrid environment

Monitoring in a hybrid environment can be a challenge. However, with tools like Azure Arc as you bring Azure services on-premises, you can easily enroll in additional Azure services such as monitoring, security, and update by simply turning them on.

- Across products: Integrate with Microsoft Sentinel, Microsoft Defender for Cloud
- Bring Microsoft Defender for Cloud to your on-prem data and servers with Arc
- Set security policies, resource boundaries, and RBAC for workloads across the hybrid infra
- Proper admin roles for read, modify, re-onboard, and delete a machine

Monitoring containers

Monitoring your containers is critical. Azure Monitor for containers provides a rich monitoring experience for the AKS and AKS engine clusters.

Configure Azure Monitor for containers to monitor Azure Arc enabled Kubernetes clusters hosted outside of Azure. This helps achieve comprehensive monitoring of your Kubernetes clusters across Azure, on-premises, and third-party cloud environments.

Azure Monitor for containers can provide you with performance visibility by collecting memory and processor metrics from controllers, nodes, and containers available in Kubernetes through the Metrics application programming interface (API). Container logs are also collected. After you enable monitoring from Kubernetes clusters, metrics and logs are automatically collected for you through a containerized version of the Log Analytics agent. Metrics are written to the metrics store and log data is written to the logs store associated with your Log Analytics workspace. For more information about Azure Monitor for containers, refer to Azure Monitor

for containers overview.

Enable Azure Monitor for containers for one or more existing deployments of Kubernetes by using either a PowerShell or a Bash script. To enable monitoring for Arc enabled Kubernetes clusters, refer to Enable monitoring of Azure Arc enabled Kubernetes cluster.

Automatically enroll in additional Azure Arc enabled resources and services. Simply turn them on when needed:

- Strengthen your security posture and protect against threats by turning on Microsoft Defender for Cloud.
- Get actionable alerts from Azure Monitor.
- Detect, investigate, and mitigate security incidents with the power of a cloud-native SIEM, by turning on Microsoft Sentinel.

Deploy and manage containerized applications

Deploy and manage containerized applications with GitHub and Azure Policy. Ensure that applications and clusters are consistently deployed and configured at scale from source control. Write with your familiar tool set to the same application service APIs that can run consistently on-premises, across multicloud, and in edge environments. Easily instrument Azure monitoring, telemetry, and security services into your hybrid apps wherever they run.

Next steps

Reliability

Reliability in a hybrid workload

11/19/2021 • 5 minutes to read • Edit Online

In the cloud, we acknowledge up front that failures will happen. Instead of trying to prevent failures altogether, the goal is to minimize the effects of a single failing component. While historically you may have purchased levels of redundant higher-end hardware to minimize the chance of an entire application platform failing, in the cloud, we acknowledge up front that failures will happen.

For hybrid scenarios, Azure offers an end-to-end backup and disaster recovery solution that's simple, secure, scalable, and cost-effective, and can be integrated with on-premises data protection solutions. In the case of service disruption or accidental deletion or corruption of data, recover your business services in a timely and orchestrated manner.

Many customers operate a second datacenter, however, Azure can help reduce the costs of deploying, monitoring, patching, and scaling on-premises disaster recovery infrastructure, without the need to manage backup resources or build a secondary datacenter.

Extend your current backup solution to Azure, or easily configure our application-aware replication and application-consistent backup that scales based on your business needs. The centralized management interface for Azure Backup and Azure Site Recovery makes it simple to define policies to natively protect, monitor, and manage enterprise workloads across hybrid and cloud. These include:

- Azure Virtual Machines
- SQL and SAP databases
- On-premises Windows servers
- VMware machines

> **NOTE**
>
> By not having to build on-premises solutions or maintain a costly secondary datacenter, customers can reduce the cost of deploying, monitoring, patching, and scaling disaster recovery infrastructure by backing up their hybrid data and applications with Azure.

Backup and Recovery

Together, Azure Backup and Azure Site Recovery use the underlying power and unlimited scale of the cloud to deliver high availability with minimal maintenance or monitoring overhead. These native capabilities are available through a pay-as-you-use model that bills only for the storage that is consumed.

Using Azure Site Recovery, users can set up and manage replication, failover, and failback from a single location in the Azure portal. The Azure hybrid services tool in Windows Admin Center can also be used as a centralized hub to easily discover all the available Azure services that bring value to on-premises or hybrid environments. Windows Admin Center streamlines setup and the process of replicating virtual machines on Hyper-V servers or clusters, making it easier to bolster the resiliency of environments with Azure Site Recovery's disaster recovery service.

Azure is committed to providing the best-in-class data protection to keep your applications running. Azure Backup protects backups of on-premises and cloud-resources from ransomware attacks by isolating backup data from source data, combined with multi-factor authentication and the ability to recover maliciously or accidentally deleted backup data. With Azure Site Recovery you can fail over VMs to the cloud or between cloud

data centers and secure them with network security groups.

In the case of a disruption, accidental deletion, or corruption of data, customers can rest assured that they will be able to recover their business services and data in a timely and orchestrated manner. These native capabilities support low recovery-point objective (RPO) and recovery-time objective (RTO) targets for any mission-critical workload in your organization. Azure is here to help customers pivot towards a strengthened BCDR strategy.

Availability Considerations

For Azure Arc

In most cases, the location you select when you create the installation script should be the Azure region geographically closest to your machine's location. The rest of the data will be stored within the Azure geography containing the region you specify, which might also affect your choice of region if you have data residency requirements. If an outage affects the Azure region to which your machine is connected, the outage will not affect the Arc enabled server, but management operations using Azure might not be able to complete. For resilience in the event of a regional outage, if you have multiple locations which provide a geographically-redundant service, it's best to connect the machines in each location to a different Azure region.

Ensure that Azure Arc is supported in your regions by checking supported regions. Also, ensure that services referenced in the Architecture section are supported in the region to which Azure Arc is deployed.

Azure Arc enabled data services

With Azure Arc enabled SQL Managed Instance, you can deploy individual databases in either a single or multiple pod pattern. For example, the developer or general-purpose pricing tier implements a single pod pattern, while a highly available business critical pricing tier implements a multiple pod pattern. A highly available Azure SQL managed instance uses Always On Availability Groups to replicate the data from one instance to another either synchronously or asynchronously.

With Azure Arc enabled SQL Managed Instance, planning for storage is also critical from the data resiliency standpoint. If there's a hardware failure, an incorrect choice might introduce the risk of total data loss. To avoid such risk, you should consider a range of factors affecting storage configuration kubernetes-storage-class-factors for both data controller and database instances.

Azure Arc enabled SQL Managed Instance provides automatic local backups, regardless of the connectivity mode. In the Directly Connected mode, you also have the option of leveraging Azure Backup for off-site, long-term backup retention.

Azure Stack HCI

Site-level fault domains

Each physical site of an Azure Stack HCI stretched cluster represents distinct fault domains that provide additional resiliency. A fault domain is a set of hardware components that share a single point of failure. To be fault tolerant to a particular level, you need multiple fault domains at that level.

Site awareness

Site awareness allows you to control placement of virtualized workloads by designating their preferred sites. Specifying the preferred site for a stretched cluster offers many benefits, including the ability to group workloads at the site level and to customize quorum voting options. By default, during a cold start, all virtual machines use the preferred site, although it is also possible to configure the preferred site at the cluster role or group level.

Resiliency limits

Azure Stack HCI provides multiple levels of resiliency, but because of its hyper-converged architecture, that resiliency is subject to limits imposed not only by the cluster quorum, but also by the pool quorum. You can

eliminate this limit by implementing cluster sets in which you combine multiple Azure Stack HCI clusters to create an HCI platform consisting of hundreds of nodes.

Next steps

Security

Security in a hybrid workload

11/19/2021 • 3 minutes to read • Edit Online

Security is one of the most important aspects of any architecture. Particularly in hybrid and multicloud environments, an architecture built on good security practices should be resilient to attacks and provide confidentiality, integrity, and availability. To assess your workload using the tenets found in the Microsoft Azure Well-Architected Framework, see the Microsoft Azure Well-Architected Review.

Microsoft Defender for Cloud can monitor on-premises systems, Azure VMs, Azure Monitor resources, and even VMs hosted by other cloud providers. To support that functionality, the standard fee-based tier of Microsoft Defender for Cloud is needed. We recommend that you use the 30-day free trial to validate your requirements. Defender for Cloud's operational process won't interfere with your normal operational procedures. Instead, it passively monitors your deployments and provides recommendations based on the security policies you enable.

Microsoft Sentinel can help simplify data collection across different sources, including Azure, on-premises solutions, and across clouds using built-in connectors. Microsoft Sentinel works to collect data at cloud scale—across all users, devices, applications, and infrastructure, both on-premises and in multiple clouds.

Azure Architecture Center (AAC) resources

- Hybrid Security Monitoring using Microsoft Defender for Cloud and Microsoft Sentinel
- DevSecOps in Azure
- Optimize administration of SQL Server instances in on-premises and multi-cloud environments by leveraging Azure Arc
- Implement a secure hybrid network
- Securely managed web applications

Principles

Azure Arc management security capabilities

- Access unique Azure security capabilities such as Microsoft Defender for Cloud.
- Centrally manage access for resources with Role-Based Access Control.
- Centrally manage and enforce compliance and simplify audit reporting with Azure Policy.

Azure Arc enabled data services security capabilities

- Protect your data workloads with Microsoft Defender for Cloud in your environment, using the advanced threat protection and vulnerability assessment features for unmatched security.
- Set security policies, resource boundaries, and role-based access control for various data workloads seamlessly across your hybrid infrastructure.

Azure Stack HCI

- **Protection in transit**. Storage Replica offers built-in security for its replication traffic. This includes packet signing, AES-128-GCM full data encryption, support for Intel AES-NI encryption acceleration, and pre-authentication integrity man-in-the-middle attack prevention.
  - Storage Replica also utilizes Kerberos AES256 for authentication between the replicating nodes.
- **Encryption at rest**. Azure Stack HCI supports BitLocker Drive Encryption for its data volumes, thus facilitating compliance with standards such as FIPS 140-2 and HIPAA.
- **Integration with a range of Azure services that provide more security advantages**. You can integrate virtualized workloads that run on Azure Stack HCI clusters with Azure services such as Microsoft

Defender for Cloud.

- **Firewall-friendly configuration**. Storage Replica traffic requires a limited number of open ports between the replicating nodes.

Design

Azure Arc enabled servers

Implement Azure Monitor

Use Azure Monitor to monitor your VMs, virtual machine scale sets, and Azure Arc machines at scale. Azure Monitor analyzes the performance and health of your Windows and Linux VMs. It also monitors their processes and dependencies on other resources and external processes. It includes support for monitoring performance and application dependencies for VMs that are hosted on-premises or in another cloud provider.

Implement Microsoft Sentinel

Use Microsoft Sentinel to deliver intelligent security analytics and threat intelligence across the enterprise. This provides a single solution for alert detection, threat visibility, proactive hunting, and threat response. Microsoft Sentinel is a scalable, cloud-native, security information event management (SIEM) and security orchestration automated response (SOAR) solution that enables several scenarios including:

- Collect data at cloud scale across all users, devices, applications, and infrastructure, both on-premises and in multiple clouds.
- Detect previously undetected threats and minimize false positives.
- Investigate threats with artificial intelligence and hunt for suspicious activities at scale.
- Respond to incidents rapidly with built-in orchestration and automation of common tasks.

Azure Stack HCI

A stretched Azure Stack HCI cluster relies on Storage Replica to perform synchronous storage replication between storage volumes hosted by the two groups of nodes in their respective physical sites. If a failure affects the availability of the primary site, the cluster automatically transitions its workloads to nodes in the surviving site to minimize potential downtime.

Monitor

- Across products: Integrate with Microsoft Sentinel and Microsoft Defender for Cloud.
- Bring Microsoft Defender for Cloud to your on-premises data and servers with Arc.
- Set security policies, resource boundaries, and RBAC for workloads across the hybrid infrastructure.
- Set the correct admin roles to read, modify, re-onboard, and delete a machine.

Azure App Service and reliability

11/19/2021 • 8 minutes to read • Edit Online

Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. This service adds the power of Microsoft Azure to your application, such as:

- Security
- Load balancing
- Autoscaling
- Automated management

To explore how Azure App Service can bolster the resiliency of your application workload, reference key features in Why use App Service?

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure App Service.

Design considerations

Microsoft guarantees that Azure App Service will be available `99.95%` of the time. However, no SLA is provided using either the Free or Shared tiers.

For more information, reference the SLA for App Service.

Checklist

Have you configured Azure App Service with resiliency in mind?

- Consider disabling ARR Affinity for your App Service.
- Use a different store for session state.
- Use Web Jobs.
- Enable Always On to ensure Web Jobs run reliably.
- Access the on-prem database using private connections like Azure VPN or Express Route.
- Set up backup and restore.
- Understand IP Address deprecation impact.
- Ensure App Service Environments (ASE) are deployed in highly available configurations across Availability Zones.
- Ensure the ASE Network is configured correctly.
- Consider configuring Upgrade preference if multiple environments are used.
- Plan for scaling out the ASE cluster.
- Use Deployment slots for resilient code deployments.
- Avoid unnecessary worker restarts.
- Use Run From Package to avoid deployment conflicts.
- Use Basic or higher plans with two or more worker instances for high availability.
- Evaluate the use of TCP and SNAT ports to avoid outbound connection errors.
- Enable Health check to identify non-responsive workers.
- Enable Autoscale to ensure adequate resources are available to service requests.
- Enable Local Cache to reduce dependencies on cluster file servers.

- Enable Diagnostic Logging to provide insight into application behavior.
- Enable Application Insights Alerts to signal fault conditions.
- Review Azure App Service diagnostics to ensure common problems are addressed.
- Evaluate per-app scaling for high density hosting on Azure App Service.

Configuration recommendations

Explore the following table of recommendations to optimize your App Service configuration for service reliability:

ASE RECOMMENDATION	DESCRIPTION
Consider disabling ARR Affinity for your App Service.	ARR Affinity (Application Request Routing) sets an affinity cookie, which is used to redirect users to the same node that handled their previous requests.
Use a different store for session state.	Storing session state in memory can result in losing session state when there's a problem with the application or App Service. It also limits the possibility of spreading the load over other instances.
Enable Always On to ensure Web Jobs run reliably.	A web app can time out after 20 minutes of inactivity. Only requests to the actual web app reset the timer. If your app runs continuously or if you schedule Timer trigger Web Jobs, enable Always On. Only available in the Basic, Standard, and Premium pricing tiers.
Access the on-prem database using private connections like Azure VPN or Express Route.	Access the on-premises database through a private connection so that the connection is secure and predictable.
Set up backup and restore.	Backup and restore lets you manually create or schedule app backups. You can retain backups for an indefinite amount of time.
Understand IP Address deprecation impact.	Floating addresses aren't guaranteed to remain on the resource. It's essential to check for deprecated IP addresses.
Deploy in highly available configuration across Availability Zones.	Ensures applications can continue to operate even if there's a datacenter-level failure. This provides excellent redundancy without requiring multiple deployments in different Azure regions.
Configure ASE Network correctly.	A common ASE pitfall occurs when ASE is deployed into a subnet with an IP address space that is too small to support future expansion. In such cases, ASE can be left unable to scale without redeploying the entire environment into a larger subnet. We highly recommend that adequate IP addresses be used to support either the maximum number of workers or the largest number considered workloads will need. A single ASE cluster can scale to `201` instances, which would require a `/24` subnet.

ASE RECOMMENDATION	DESCRIPTION
Configure **Upgrade preference** if multiple environments are used.	If lower environments are used for staging or testing, consider configuring these environments to receive updates sooner than the production environment. This will help to identify any conflicts or problems with an update, and provides a window to mitigate issues before they reach the production environment. If multiple load balanced (zonal) production deployments are used, *Upgrade preference* can be used to protect the broader environment against issues from platform upgrades.
Scale out the ASE cluster.	Scaling ASE instances vertically or horizontally takes `30` to `60` minutes as new private instances need to be provisioned. We highly recommend investing in up-front planning for scaling during spikes in load or transient failure scenarios.
Use **Deployment slots** for resilient code deployments.	*Deployment slots* allow for code to be deployed to instances that are *warmed-up* before serving production traffic. For more information, reference Testing in production with Azure App Service.
Avoid unnecessary worker restarts.	Many events can lead App Service workers to restart, such as content deployment, App Settings changes, and VNet integration configuration changes. A best practice is to make changes in a deployment slot other than the slot currently configured to accept production traffic. After workers are recycled and warmed up, a *swap* can be performed without unnecessary down time.
`Run From Package` to avoid deployment conflicts	`Run from Package` provides several advantages: - Eliminates file lock conflicts between deployment and runtime. - Ensures only fully deployed apps are running at any time. - May reduce *cold-start* times, particularly for JavaScript functions with large `npm` package trees.
Use Basic or higher plans with two or more worker instances for high availability.	Azure App Service provides many configuration options that aren't enabled by default.
Evaluate the use of TCP and SNAT ports.	TCP connections are used for all outbound connections; but, SNAT ports are used when making outbound connections to public IP addresses. SNAT port exhaustion is a common failure scenario that can be predicted by load testing while monitoring ports using Azure Diagnostics. For more information, reference TCP and SNAT ports.
Enable **Health check** to identify non-responsive workers.	Any health check is better than none at all. The logic behind endpoint tests should assess all critical downstream dependencies to ensure overall health. As a best practice, we highly recommend tracking application health and cache status in real time as this removes unnecessary delays before action can be taken.

ASE RECOMMENDATION	DESCRIPTION
Enable **Autoscale** to ensure adequate resources are available to service requests.	The default limit of App Service workers is `30`. If the App Service routinely uses `15` or more instances, consider opening a support ticket to increase the maximum number of workers to `2x` the instance count required to serve normal peak load.
Enable `Local_Cache` to reduce dependencies on cluster file servers.	Enabling local cache is always appropriate because it can lead to slower worker startup times. When coupled with **Deployment slots**, it can improve resiliency by removing dependencies on file servers and also reduces storage-related recycle events. Don't use local cache with a single worker instance or when shared storage is required.
Enable **Diagnostic Logging** to provide insight into application behavior.	*Diagnostic logging* provides the ability to ingest rich application and platform-level logs through Log Analytics, Azure Storage, or a third-party tool using Event Hub.
Enable **Application Insights alerts** to make you aware of fault conditions.	Application performance monitoring with Application Insights provides deep analyses into application performance. For Windows Plans, a *codeless deployment* approach is possible to quickly get a performance analysis without changing any code.
Review **Azure App Service diagnostics** to ensure common problems are addressed.	It's a good practice to regularly review service-related diagnostics and recommendations, and take action as appropriate.
Evaluate **per-app scaling** for high density hosting on Azure App Service.	Per-app scaling can be enabled at the App Service plan level to allow for scaling an app independently from the App Service plan that hosts it. This way, an App Service plan can be scaled to `10` instances, but an app can be set to use only five. Apps are allocated within the available App Service plan using a best effort approach for even distribution across instances. While an even distribution isn't guaranteed, the platform will make sure that two instances of the same app won't be hosted on the same App Service plan instance.

TCP and SNAT ports

If a load test results in SNAT errors, it's necessary to either scale across more or larger workers, or implement coding practices to help preserve and reuse SNAT ports, such as connection pooling and the lazy loading of resources. We don't recommend exceeding `100` simultaneous outbound connections to a public IP address per worker, and to avoid communicating with downstream services through public IP addresses when a private address (Private Endpoint) or Service Endpoint through vNet Integration could be used. TCP port exhaustion happens when the sum of connection from a given worker exceeds the capacity. The number of available TCP ports depend on the size of the worker.

The following table lists the current limits:

TCP PORTS	SMALL (B1, S1, P1, I1)	MEDIUM (B2, S2, P2, I2)	LARGE (B3, S3, P3, I3)
TCP ports	1920	3968	8064

Applications with many longstanding connections require ports to be left open for long periods of time, which can lead to TCP Connection exhaustion. TCP Connection limits are fixed based on instance size, so it's necessary

to scale up to a larger worker size to increase the allotment of TCP connections, or implement code level mitigations to govern connection usage. Similar to SNAT port exhaustion, you can use Azure Diagnostics to identify a problem exists with TCP port limits.

Source artifacts

To identify App Service Plans with only one instance, use the following query:

```
Resources
| where type == "microsoft.web/serverfarms" and properties.computeMode == `Dedicated`
| where sku.capacity == 1
```

Learn more

The Ultimate Guide to Running Healthy Apps in the Cloud

Next step

Azure App Service and cost optimization

Azure App Service and cost optimization

11/19/2021 • 2 minutes to read • Edit Online

Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. This service adds the power of Microsoft Azure to your application, such as:

- Security
- Load balancing
- Autoscaling
- Automated management

To explore how to optimize costs for Azure App Service in your workload, reference key features in Why use App Service?

The following sections include a checklist and recommended configuration options specific to Azure App Service.

Checklist

Have you configured Azure App Service while considering cost optimization?

- Ensure the ASE subnet is appropriately sized.
- Consider cost savings by using the App Service Premium v3 plan over the Premium v2 plan.
- Always use a scale-out and scale-in rule combination.
- Understand the behavior of multiple scaling rules in a profile.
- Consider Basic or Free tier for non-production usage.

Configuration recommendations

Explore the following table of recommendations to optimize your App Service configuration for service cost:

ASE RECOMMENDATION	DESCRIPTION
Ensure the ASE subnet is appropriately sized.	The size of the subnet used to host an ASE directly affects maximum scale. An ASE with no App Service plans will use `12` to `13` addresses before you create an app. It's recommended that you deploy ASEs into a `/24` subnet. The maximum number of nodes in an ASE is `100`. During a scale-up event, the new machines are provisioned and placed into the subnet before the applications are migrated to the new machines, and the old machines are removed. The subnet must allow for at least `200` machines to handle the maximum deployment size, which requires a `/24` subnet. If you plan for insufficient capacity, scale-out operations will be limited.
Use App Service Premium v3 plan over the Premium v2 plan	The App Service Premium (v3) Plan has a `20%` discount versus comparable Pv2 configurations. Reserved Instance commitment (1Y, 3Y, Dev/Test) discounts are available for App Services running in the Premium v3 plan.

ASE RECOMMENDATION	DESCRIPTION
Use a scale-out and scale-in rule combination	If you use only one part of the combination, autoscale will only take action in a single direction (scale out, or in) until it reaches the maximum, or minimum instance counts defined in the profile. This scaling behavior isn't optimal, ideally you want your resource to scale up at times of high usage to ensure availability. Similarly, at times of low usage, you want your resource to scale down, so you can realize cost savings.
Understand the behavior of multiple scaling rules in a profile.	There are cases where you may have to set multiple rules in a profile. On scale-out, autoscale runs if `any` rule is met. On scale-in, autoscale requires `all` rules to be met.
Consider Basic or Free tier for non-production usage.	For non-prod App Service plans, consider scaling to Basic or Free Tier and scale up, as needed, and scale down when not in use – for example, during a Load Test exercise or based on the capabilities provided (such as custom domain, SSL, and more).

Next step

Azure App Service and operational excellence

Azure App Service and operational excellence

11/19/2021 • 7 minutes to read • Edit Online

Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. This service adds the power of Microsoft Azure to your application, such as:

- Security
- Load balancing
- Autoscaling
- Automated management

To explore how Azure App Service can benefit the operational excellence of your application workload, reference key features in Why use App Service?

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure App Service.

Design considerations

Microsoft guarantees that Azure App Service will be available `99.95%` of the time. However, no SLA is provided using either the Free or Shared tiers.

For more information, reference the SLA for App Service.

Checklist

Have you configured Azure App Service while considering operational excellence?

- Create a deployment plan because redeploying the app service can reset the scaled units.
- Review the App Service Advisor recommendations.
- Ensure you configure the App Service Environments (ASE) Network correctly.
- Consider configuring Upgrade Preference if you're using multiple environments.
- Plan for scaling out the ASE cluster.
- Use Deployment Slots for resilient code deployments.
- Avoid unnecessary worker restarts when deploying application code or configuration.
- Use Run From Package to avoid deployment conflicts.
- Use Basic or higher plans with two or more worker instances for high availability.
- Evaluate the use of TCP and SNAT ports to avoid outbound connection errors.
- Enable Health check to identify non-responsive workers.
- Enable Autoscale to ensure adequate resources are available to service requests.
- Enable Local Cache to reduce dependencies on cluster file servers.
- Enable Diagnostic Logging to provide insight into application behavior.
- Enable Application Insights Alerts to signal fault conditions.
- Review Azure App Service diagnostics to ensure common problems are addressed.
- Evaluate per-app scaling for high density hosting on Azure App Service.

ASE RECOMMENDATION	DESCRIPTION
Create a deployment plan because redeploying the app service can reset the scaled units.	Automatic scaling rules apply during operation of the environment, but redeploying the app service may cause the plan to reset to the default number of units. Customers should be aware of this behavior and plan for it during deployments. Deploy only during off-peak times or deploy maximum units with automatic scaling enabled to scale in and out to prevent website performance implications.
Review the App Service Advisor recommendations.	App Service Advisor gives you real-time recommendations in the portal on resource exhaustion and conditions related to CPU, memory, and connections.
Ensure you configure the **App Service Environments (ASE) Network** correctly.	One common ASE pitfall occurs when ASE is deployed into a subnet with an IP address space that is too small to support future expansion. In such cases, ASE can be left unable to scale without redeploying the entire environment into a larger subnet. We highly recommended that adequate IP addresses be used to support either the maximum number of workers or the largest number considered workloads will need. A single ASE cluster can scale to `201` instance, which would require a `/24` subnet.
Configure **Upgrade preference** if you're using multiple environments.	If lower environments are used for staging or testing, consider configuring these environments to receive updates sooner than the production environment. This will help to identify any conflicts or problems with an update, and provides a window to mitigate issues before they reach the production environment. If multiple load balanced (zonal) production deployments are used, *Upgrade preference* can be used to protect the broader environment against issues from platform upgrades.
Scale out the ASE cluster.	Scaling ASE instances vertically or horizontally takes `30` to `60` minutes as new private instances need to be provisioned. We highly recommend investing in up-front planning for scaling during spikes in load or transient failure scenarios.
Use **Deployment slots** for resilient code deployments.	*Deployment slots* allow for code to be deployed to instances that are *warmed-up* before serving production traffic. For more information, reference Testing in production with Azure App Service.
Avoid unnecessary worker restarts.	Many events can lead App Service workers to restart, such as content deployment, App Settings changes, and VNet integration configuration changes. A best practice is to make changes in a deployment slot other than the slot currently configured to accept production traffic. After workers are recycled and warmed up, a *swap* can be performed without unnecessary down time.
`Run From Package` to avoid deployment conflicts	`Run from Package` provides several advantages: - Eliminates file lock conflicts between deployment and runtime. - Ensures only fully deployed apps are running at any time. - May reduce *cold-start* times, particularly for JavaScript functions with large `npm` package trees.

ASE RECOMMENDATION	DESCRIPTION
Use Basic or higher plans with two or more worker instances for high availability.	Azure App Service provides many configuration options that aren't enabled by default.
Evaluate the use of TCP and SNAT ports.	TCP connections are used for all outbound connections, but SNAT ports are used when making outbound connections to public IP addresses. SNAT port exhaustion is a common failure scenario that can be predicted by load testing while monitoring ports using Azure Diagnostics. For more information, reference TCP and SNAT ports.
Enable **Health check** to identify non-responsive workers.	Any health check is better than none at all. The logic behind endpoint tests should assess all critical downstream dependencies to ensure overall health. As a best practice, we highly recommend tracking application health and cache status in real time as this removes unnecessary delays before action can be taken.
Enable **Autoscale** to ensure adequate resources are available to service requests.	The default limit of App Service workers is `30`. If the App Service routinely uses `15` or more instances, consider opening a support ticket to increase the maximum number of workers to `2x` the instance count required to serve normal peak load.
Enable `Local_Cache` to reduce dependencies on cluster file servers.	Enabling local cache is always appropriate because it can lead to slower worker startup times. When coupled with **Deployment slots**, it can improve resiliency by removing dependencies on file servers and also reduces storage-related recycle events. Don't use local cache with a single worker instance or when shared storage is required.
Enable **Diagnostic Logging** to provide insight into application behavior.	*Diagnostic logging* provides the ability to ingest rich application and platform-level logs through Log Analytics, Azure Storage, or a third-party tool using Event Hub.
Enable **Application Insights alerts** to make you aware of fault conditions.	Application performance monitoring with Application Insights provides deep analyses into application performance. For Windows Plans, a *codeless deployment* approach is possible to quickly get a performance analysis without changing any code.
Review **Azure App Service diagnostics** to ensure common problems are addressed.	It's a good practice to regularly review service-related diagnostics and recommendations, and take action as appropriate.
Evaluate **per-app scaling** for high density hosting on Azure App Service.	Per-app scaling can be enabled at the App Service plan level to allow for scaling an app independently from the App Service plan that hosts it. This way, an App Service plan can be scaled to `10` instances, but an app can be set to use only five. Apps are allocated to available App Service plan using a best effort approach for an even distribution across instances. While an even distribution isn't guaranteed, the platform will make sure that two instances of the same app won't be hosted on the same App Service plan instance.

TCP and SNAT ports

If a load test results in SNAT errors, it's necessary to either scale across more or larger workers, or implement coding practices to help preserve and reuse SNAT ports, such as connection pooling and the lazy loading of

resources. We don't recommend exceeding `100` simultaneous outbound connections to a public IP address per worker, and to avoid communicating with downstream services through public IP addresses when a private address (Private Endpoint) or Service Endpoint through vNet Integration could be used. TCP port exhaustion happens when the sum of connection from a given worker exceeds the capacity. The number of available TCP ports depend on the size of the worker.

The following table lists the current limits:

TCP PORTS	SMALL (B1, S1, P1, I1)	MEDIUM (B2, S2, P2, I2)	LARGE (B3, S3, P3, I3)
TCP ports	1920	3968	8064

Applications with many longstanding connections require ports to be left open for long periods of time, which can lead to TCP Connection exhaustion. TCP Connection limits are fixed based on instance size, so it's necessary to scale up to a larger worker size to increase the allotment of TCP connections, or implement code level mitigations to govern connection usage. Similar to SNAT port exhaustion, you can use Azure Diagnostics to identify if a problem exists with TCP port limits.

Source artifacts

To identify App Service plans with only one instance, use the following query:

```
Resources
| where type == "microsoft.web/serverfarms" and properties.computeMode == `Dedicated`
| where sku.capacity == 1
```

Learn more

The Ultimate Guide to Running Healthy Apps in the Cloud

Next step

Azure Batch and reliability

Azure Batch and reliability

11/19/2021 • 2 minutes to read • Edit Online

Azure Batch allows you to run large-scale parallel and high-performance computing (HPC) batch jobs efficiently in Azure.

Use Azure Batch to:

- Create and manage a pool of compute nodes (virtual machines).
- Install applications you want to run.
- Schedule jobs to run on the compute nodes.

The following sections include a design and configuration checklist, recommended design, and configuration options specific to Azure Batch.

Design and configuration checklist

Have you designed your workload and configured Azure Batch with resiliency in mind?

- Keep application binaries and reference data up to date in all regions.
- Use fewer jobs and more tasks.
- Use multiple Batch accounts in various regions to allow your application to continue running, if an Azure Batch account in one region becomes unavailable.
- Build durable tasks.
- Pre-create all required services in each region, such as the Batch account and storage account.
- Make sure the appropriate quotas are set on all subscriptions ahead of time, so you can allocate the required number of cores using the Batch account.

Design and configuration recommendations

Explore the following table of recommendations to optimize your workload design and Azure Batch configuration for service reliability:

RECOMMENDATION	DESCRIPTION
Keep application binaries and reference data up to date in all regions.	Staying up to date will ensure the region can be brought online quickly without waiting for file upload and deployment.
Use fewer jobs and more tasks.	Using a job to run a single task is inefficient. For example, it's more efficient to use a single job containing `1000` tasks rather than creating `100` jobs that contain `10` tasks each. Running `1000` jobs, each with a single task, would be the least efficient, slowest, and most expensive approach.
Use multiple Batch accounts in various regions to allow your application to continue running, if an Azure Batch account in one region becomes unavailable.	It's crucial to have multiple accounts for a highly available application.

RECOMMENDATION	DESCRIPTION
Build durable tasks.	Tasks should be designed to withstand failure and accommodate retry, especially for long running tasks. Ensure tasks generate the same, single result even if they're run more than once. One way to achieve the same result is to make your tasks *goal seeking*. Another way is to make sure your tasks are *idempotent* (tasks will have the same outcome no matter how many times they're run).
Pre-create all required services in each region, such as the Batch account and storage account.	There's often no charge for creating accounts and charges accrue only when you use the account, or when you store data.

Next step

Azure Batch and operational excellence

Azure Batch and operational excellence

11/19/2021 • 2 minutes to read • Edit Online

Azure Batch allows you to run large-scale parallel and high-performance computing (HPC) batch jobs efficiently in Azure.

Use Azure Batch to:

- Create and manage a pool of compute nodes (virtual machines).
- Install applications you want to run.
- Schedule jobs to run on the compute nodes.

The following sections include a design and configuration checklist, recommended design, and configuration options specific to Azure Batch.

Design and configuration checklist

Have you designed your workload and configured Azure Batch with operational excellence in mind?

- Keep application binaries and reference data up to date in all regions.
- Use fewer jobs and more tasks.
- Pre-create all required services in each region, such as the Batch account and storage account.
- Make sure the appropriate quotas are set on all subscriptions ahead of time, so you can allocate the required number of cores using the Batch account.

Design and configuration recommendations

Explore the following table of recommendations to optimize your workload design and Azure Batch configuration for operational excellence:

RECOMMENDATION	DESCRIPTION
Keep application binaries and reference data up to date in all regions.	Staying up to date will ensure the region can be brought online quickly without waiting for file upload and deployment.
Use fewer jobs and more tasks.	Using a job to run a single task is inefficient. For example, it's more efficient to use a single job containing `1000` tasks rather than creating `100` jobs that contain `10` tasks each. Running `1000` jobs, each with a single task, would be the least efficient, slowest, and most expensive approach.
Pre-create all required services in each region, such as the Batch account and storage account.	There's often no charge for creating accounts and charges accrue only when you use the account, or when you store data.

Next step

Azure Batch and performance efficiency

Azure Batch and performance efficiency

11/19/2021 • 2 minutes to read • Edit Online

Azure Batch allows you to run large-scale parallel and high-performance computing (HPC) batch jobs efficiently in Azure.

Use Azure Batch to:

- Create and manage a pool of compute nodes (virtual machines).
- Install applications you want to run.
- Schedule jobs to run on the compute nodes.

The following sections include a design checklist and recommended design options specific to Azure Batch.

Design checklist

Have you designed your workload and configured Azure Batch with performance efficiency in mind?

- Use fewer jobs and more tasks.

Design and configuration recommendations

Consider the following recommendation to optimize your workload design and Azure Batch configuration for performance efficiency:

RECOMMENDATION	DESCRIPTION
Use fewer jobs and more tasks.	Using a job to run a single task is inefficient. For example, it's more efficient to use a single job containing `1000` tasks rather than creating `100` jobs that contain `10` tasks each. Running `1000` jobs, each with a single task, would be the least efficient, slowest, and most expensive approach.

Next step

AKS and reliability

Azure Kubernetes Service (AKS) and reliability

11/19/2021 • 9 minutes to read • Edit Online

Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance.

To explore how AKS can bolster the resiliency of your application workload, reference architecture guidance and best practices on the AKS Solution architectures page.

The following sections include configuration checklists, recommended configuration options, and supporting source artifacts specific to AKS.

Checklists

Have you configured Azure Kubernetes Service (AKS) with resiliency in mind?

- Run resource-intensive applications, such as ingress controllers, or workloads that require specialized hardware (such as large data processing workloads like machine learning (ML) or artificial intelligence (AI)).
- Don't modify resources in the node resource group (for example MC_). You should *only* make modifications at cluster creation time, or with assistance from Azure Support.
- Define Pod resource requests and limits in application deployment manifests.
- Control pod scheduling using node selectors and affinity.
- Use a central monitoring tool, such as Azure Monitor and App Insights to centrally collect metrics, logs, and diagnostics for troubleshooting purposes.
- Enable and review Kubernetes master node logs.
- Configure scraping of Prometheus metrics with Azure Monitor for containers.
- Use Azure Network Policies or Calico to control traffic between pods. *Requires CNI Network Plug-in*.
- Ensure proper selection of network plugin based on network requirements and cluster sizing.
- Use the AKS Uptime SLA for production grade clusters.
- Store container images within Azure Container Registry and enable geo-replication to replicate container images across leveraged AKS regions.
- Enable Azure Defender for container registries to enable vulnerability scanning for container images.
- Authenticate with Azure AD to Azure Container Registry.
- Use Availability Zones to maximize resilience within an Azure region by distributing AKS agent nodes across physically separate data centers.
- Use a template-based deployment using Terraform, Ansible, and others only. Make sure that all deployments are repeatable and traceable, and stored in a source code repo. Can be combined with GitOps.
- Adopt a multiregion strategy by deploying AKS clusters deployed across different Azure regions to maximize availability and provide business continuity.

Node pool design checklist

- Use Virtual Machine Scale Set (VMSS) VM set type for AKS node pools.
- Keep the System node pool isolated from application workloads.
- Use dedicated node pools for infrastructure tools that require high resource utilization, such as, Istio, or have a special scale, or load behavior.
- Separate applications to dedicated node pools based on specific requirements.

- Use taints and tolerations to provide dedicated nodes and limit resource intensive applications.
- Consider the use of Virtual Nodes (vKubelet) with ACI for rapid, massive, and infinite scale.

Scalability checklist

- Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.
- Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.
- Use the Horizontal pod autoscaler to adjust the number of pods in a deployment depending on CPU utilization or other select metrics.
- Separate workloads into different node pools and consider scaling user node pools to zero.

AKS roadmap and GitHub release notes checklist

- Subscribe to the public AKS Roadmap Release Notes on GitHub to stay up to date on upcoming changes, improvements, Kubernetes version releases, and the deprecation of old releases.
- Regularly upgrade to a supported version of Kubernetes.
- Regularly process node image updates.
- Leverage AKS Cluster auto-upgrade with Planned Maintenance.

Security guidelines checklist

- Use Managed Identities to avoid managing and rotating service principles.
- Use AAD integration to take advantage of centralized account management, passwords, application access management, and identity protection.
- Use Kubernetes RBAC with AAD for least privilege and minimize granting administrator privileges to protect configuration, and secrets access.
- Limit access to Kubernetes cluster configuration file with Azure role-based access control.
- Limit access to actions that containers can perform. Provide the least number of permissions, and avoid the use of root, or privileged escalation.
- Evaluate the use of the built-in AppArmor security module to limit actions that containers can perform such as read, write, or execute, or system functions such as mounting files systems.
- Evaluate the use of the seccomp (secure computing). Seccomp works at the process level and allows you to limit the process calls that containers can perform.
- Use Pod Identities and Secrets Store CSI Driver with Azure Key Vault to protect secrets, certificates, and connection strings.
- Use Azure Security Center to provide AKS recommendations.
- Secure clusters and pods with Azure Policy.

AKS configuration recommendations

Explore the following table of recommendations to optimize your AKS configuration for service reliability:

AKS RECOMMENDATION	DESCRIPTION
Control pod scheduling using node selectors and affinity.	Allows the Kubernetes scheduler to logically isolate workloads by hardware in the node. Unlike tolerations, pods without a matching node selector can be scheduled on labeled nodes, which allows unused resources on the nodes to consume, but gives priority to pods that define the matching node selector. Use node affinity for more flexibility, which allows you to define what happens if the pod can't be matched with a node.

AKS RECOMMENDATION	DESCRIPTION
Configure scraping of Prometheus metrics with Azure Monitor for containers.	Azure Monitor for containers provides a seamless onboarding experience to collect Prometheus metrics. Reference Configure scraping of Prometheus metrics with Azure Monitor for containers for more information.
Ensure proper selection of network plugin based on network requirements and cluster sizing.	Azure CNI is required for specific scenarios, for example, Windows-based node pools, specific networking requirements and Kubernetes Network Policies. Reference Kubenet vs. Azure CNI for more information.
Use the AKS Uptime SLA for production grade clusters.	The AKS Uptime SLA guarantees: - `99.95%` availability of the Kubernetes API server endpoint for AKS Clusters that use Azure Availability Zones, or - `99.9%` availability for AKS Clusters that don't use Azure Availability Zones.
Authenticate with Azure AD to Azure Container Registry.	AKS and Azure AD enables authentication with Azure Container Registry without the use of K8s and `imagePullSecrets` secrets. Reference Authenticate with Azure Container Registry from Azure Kubernetes Service for more information.
Use Availability Zones to maximize resilience within an Azure region by distributing AKS agent nodes across physically separate data centers.	Where colocality requirements exist, either a regular VMSS-based AKS deployment into a single zone or proximity placement groups can be used to minimize internode latency.
Adopt a multiregion strategy by deploying AKS clusters deployed across different Azure regions to maximize availability and provide business continuity.	Internet facing workloads should leverage Azure Front Door, Azure Traffic Manager, or a third-party CDN to route traffic globally across AKS clusters.

Node pool design recommendations

The following table reflects node pool design recommendations and descriptions related to the overall AKS configuration recommendations:

NODE POOL DESIGN RECOMMENDATIONS	DESCRIPTION
Keep the System node pool isolated from application workloads.	System node pools require a VM SKU of at least `2` `vCPUs` and `4GB` memory. Reference System and user node pools for detailed requirements.
Separate applications to dedicated node pools based on specific requirements.	Such as GPU, high memory VMs, scale-to-zero, Spot VMs, and so on. Avoid large numbers of node pools to reduce extra management overhead.

Scalability recommendations

The following table reflects scalability recommendations and descriptions related to the overall AKS configuration recommendations:

SCALABILITY RECOMMENDATIONS	DESCRIPTION
Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.	The ability to automatically scale up or down the number of nodes in your AKS cluster lets you run an efficient, cost-effective cluster.

SCALABILITY RECOMMENDATIONS	DESCRIPTION
Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.	For example, workloads such as batch processing jobs, development, testing environments, and large compute workloads may be good candidates to be scheduled on a spot node pool. Using spot VMs for nodes with your AKS cluster allows you to take advantage of unused capacity in Azure at a significant cost savings.
Separate workloads into different node pools and consider scaling user node pools to zero.	Unlike System node pools that always require running nodes, user node pools allow you to scale to `0`.

AKS roadmap and GitHub release notes recommendations

The following table reflects AKS roadmap and GitHub release notes recommendations, and descriptions related to the overall AKS configuration recommendations:

ROADMAP AND RELEASE NOTES RECOMMENDATIONS	DESCRIPTION
Subscribe to the AKS Roadmap and Release Notes on GitHub	Make sure that you're subscribed to the public AKS Roadmap Release Notes on GitHub to stay up to date on upcoming changes, improvements, and most importantly Kubernetes version releases, and the deprecation of old releases.
Regularly upgrade to a supported version of Kubernetes.	AKS supports three minor versions of Kubernetes. When a new minor patch version is introduced, the oldest minor version and patch releases supported are retired. Minor updates to Kubernetes happen on a periodic basis. It's important to have a governance process to check and upgrade as needed to not fall out of support. For more information, reference Supported Kubernetes versions AKS.
Regularly process node image updates.	AKS supports upgrading the images on a node to be up to date with the newest OS and runtime updates without updating the version of Kubernetes. The AKS team provides one new image version per week with the latest updates, including Linux or Windows patches.
Leverage AKS Cluster auto-upgrade with Planned Maintenance.	AKS supports different auto-upgrade channels (08/18/21 in public preview) to upgrade AKS clusters to newer versions of Kubernetes and newer node images once available. Planned Maintenance (08/18/21 in public preview) can be used to define maintenance windows for these operations.

Security guideline recommendations

The following table reflects security guideline recommendations and descriptions related to the overall AKS configuration recommendations:

SECURITY GUIDELINE RECOMMENDATIONS	DESCRIPTION

SECURITY GUIDELINE RECOMMENDATIONS	DESCRIPTION
Secure clusters and pods with Azure Policy.	Azure Policy can help to apply at-scale enforcements and safeguards on your clusters in a centralized, consistent manner. It can also control what functions pods are granted and if anything is running against company policy. This access is defined through built-in policies provided by the Azure Policy Add-on for AKS. By providing more control over the security aspects of your pod's specification, like root privileges, Azure Policy enables stricter security adherence and visibility into what you've deployed in your cluster. If a pod doesn't meet conditions specified in the policy, Azure Policy can disallow the pod to start or flag a violation.

Source artifacts

To identify which networking plugin (CNI or Kubenet) that AKS clusters are using, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| project name, location, resourceGroup, subscriptionId, properties.networkProfile.networkPlugin
```

To identify AKS clusters *not* deployed across **Availability Zones**, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| where isnull(zones)
```

To identify AKS clusters deployed using **Availability Sets**, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| where properties.agentPoolProfiles[0].type != 'VirtualMachineScaleSets'
| project name, location, resourceGroup, subscriptionId, properties.agentPoolProfiles[0].type
```

To identify AKS clusters *not* using **RBAC**, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| where properties.enableRBAC == false
```

To identify AKS clusters *not* deployed using a **Managed Identity**, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| where isnull(identity)
```

Next step

AKS and security

Azure Kubernetes Service (AKS) and security

11/19/2021 • 7 minutes to read • Edit Online

Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance.

To explore how AKS can bolster the security of your application workload, reference architecture guidance and best practices on the AKS Solution architectures page.

The following sections include configuration checklists, recommended configuration options, and supporting source artifacts specific to AKS.

Checklists

Have you configured Azure Kubernetes Service (AKS) with security in mind?

- Define Pod resource requests and limits in application deployment manifests.
- Use a central monitoring tool, such as Azure Monitor and App Insights, to centrally collect metrics, logs, and diagnostics for troubleshooting purposes.
- Enable and review Kubernetes master node logs.
- Configure scraping of Prometheus metrics with Azure Monitor for containers.
- Use Azure Network Policies or Calico to control traffic between pods. *Requires CNI Network Plug-in*.
- Ensure proper selection of network plugin based on network requirements and cluster sizing.

Security guidelines checklist

- Use namespaces to logically group workloads within the cluster.
- Use Managed Identities to avoid managing and rotating service principles.
- Use AAD integration to take advantage of centralized account management, passwords, application access management, and identity protection.
- Use Kubernetes RBAC with AAD for least privilege and minimize granting administrator privileges to protect configuration, and secrets access.
- Limit access to Kubernetes cluster configuration file with Azure role-based access control.
- Limit access to actions that containers can perform. Provide the least number of permissions, and avoid the use of root, or privileged escalation.
- Evaluate the use of the built-in AppArmor security module to limit actions that containers can perform such as read, write, or execute, or system functions such as mounting files systems.
- Evaluate the use of the seccomp (secure computing). Seccomp works at the process level and allows you to limit the process calls that containers can perform.
- Use Pod Identities and Secrets Store CSI Driver with Azure Key Vault to protect secrets, certificates, and connection strings.
- Protect the API server with Azure Active Directory RBAC.
- Use Azure Security Center to provide AKS recommendations.
- Secure clusters and pods with Azure Policy.
- Secure container access to resources.
- Upgrade Kubernetes clusters regularly to stay current.
- When using Azure CNI, create planned IP ranges to allow your cluster to scale in network.

- Use ingress controllers for distributing traffic.
- Use a Web Application Firewall to secure traffic.
- Use Kubernetes network policies for controlling traffic flow between pods.
- Securely connect to Kubernetes nodes through a bastion host.

AKS configuration recommendations

Explore the following table of recommendations to optimize your AKS configuration for service security:

AKS RECOMMENDATION	DESCRIPTION
Configure scraping of Prometheus metrics with Azure Monitor for containers.	Azure Monitor for containers provides a seamless onboarding experience to collect Prometheus metrics. Reference Configure scraping of Prometheus metrics with Azure Monitor for containers for more information.
Ensure proper selection of network plugin based on network requirements and cluster sizing.	Azure CNI is required for specific scenarios, for example, Windows-based node pools, specific networking requirements, and Kubernetes Network Policies. Reference Kubenet vs. Azure CNI for more information.

Security guideline recommendations

The following table reflects security guideline recommendations and descriptions related to the overall AKS configuration recommendations:

SECURITY GUIDELINE RECOMMENDATIONS	DESCRIPTION
Use AAD integration.	Using Azure AD centralizes the identity management component. Any change in user account or group status is automatically updated in access to the AKS cluster. The developers and application owners of your Kubernetes cluster need access to different resources. Kubernetes doesn't provide an identity management solution to control which users can interact with what resources. You typically integrate your cluster with an existing identity solution. Azure AAD provides an enterprise-ready identity management solution, and can integrate with AKS clusters.
Use pod identities.	Don't use fixed credentials within pods or container images because they're at risk of exposure or abuse. Instead, use pod identities to request access using a central Azure AD identity solution. *Note: Pod identities are intended for use with Linux pods and container images only.*
Protect the API server with Azure Active Directory RBAC.	Securing access to the Kubernetes API Server is one of the most important things you can do to secure your cluster. Integrate Kubernetes role-based access control (RBAC) with Azure AD to control access to the API server. These controls let you secure AKS the same way that you secure access to your Azure subscriptions. The Kubernetes API server provides a single connection point for requests to perform actions within a cluster. To secure and audit access to the API server, limit access and provide the least-privileged access permissions required. This approach isn't unique to Kubernetes, but is especially important when the AKS cluster is logically isolated for multi-tenant use.

SECURITY GUIDELINE RECOMMENDATIONS	DESCRIPTION
Secure clusters and pods with Azure Policy.	Azure Policy can help to apply at-scale enforcements and safeguards on your clusters in a centralized, consistent manner. It can also control what functions pods are granted and if anything is running against company policy. This access is defined through built-in policies provided by the Azure Policy Add-on for AKS. By providing more control over the security aspects of your pod's specification, like root privileges, Azure Policy enables stricter security adherence and visibility into what you've deployed in your cluster. If a pod doesn't meet conditions specified in the policy, Azure Policy can disallow the pod to start or flag a violation.
Secure container access to resources.	Limit access to actions that containers can perform. Provide the least number of permissions, and avoid the use of root or privileged escalation. In the same way that you should grant users or groups the least number of privileges required, containers should also be limited to only the actions and process that they need. To minimize the risk of attack, don't configure applications and containers that require escalated privileges or root access.
Upgrade Kubernetes clusters regularly to stay current.	To stay current on new features and bug fixes, regularly upgrade to the Kubernetes version in your AKS cluster. Kubernetes releases new features at a quicker pace than more traditional infrastructure platforms. Kubernetes updates include new features, bug, or security fixes. New features typically move through an alpha and then beta status before they become stable, and are generally available, and recommended for production use. This release cycle should allow you to update Kubernetes without regularly encountering breaking changes, or adjusting your deployments and templates.
Create planned IP ranges to allow your cluster to scale in network when using Azure CNI.	Clusters configured with Azure CNI networking require additional planning. The size of your virtual network and its subnet must accommodate the number of pods you plan to run and the number of nodes for the cluster.
Use ingress controllers for distributing traffic.	To distribute HTTP or HTTPS traffic to your applications, use ingress resources and controllers to provide extra features over a regular Azure load balancer, and can be managed as native Kubernetes resources.
Use a Web Application Firewall to secure traffic.	To scan incoming traffic for potential attacks, use a web application firewall such as Barracuda WAF for Azure or Azure Application Gateway. An ingress controller that distributes traffic to services and applications is typically a Kubernetes resource in your AKS cluster. In larger environments, you often want to offload some of this traffic routing or TLS termination to a network resource outside of the AKS cluster. You also want to scan incoming traffic for potential attacks.
Use Kubernetes network policies for controlling traffic flow between pods.	Use network policies to allow or deny traffic to pods. By default, all traffic is allowed between pods within a cluster. For improved security, define rules that limit pod communication. *Note: Network policy should only be used for Linux-based nodes and pods in AKS.*

SECURITY GUIDELINE RECOMMENDATIONS	DESCRIPTION
Securely connect to Kubernetes nodes through a bastion host.	Don't expose remote connectivity to your AKS nodes. Create a bastion host, or jump box, in a management virtual network. Use the bastion host to securely route traffic into your AKS cluster to remote management tasks. *Note: Most operations in AKS can be completed using the Azure management tools or though the Kubernetes API server.* AKS nodes aren't connected to the public internet, and are only available on a private network. To connect to nodes and perform maintenance or troubleshoot issues, route your connections through a bastion host, or jump box.

Source artifacts

To identify AKS clusters *not* using **RBAC**, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| where properties.enableRBAC == false
```

Azure Kubernetes Service (AKS) and cost optimization

11/19/2021 • 3 minutes to read • Edit Online

Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance.

To explore how AKS can bolster cost optimization of your application workload, reference architecture guidance and best practices on the AKS Solution architectures page.

The following sections include configuration checklists and recommended configuration options specific to AKS.

Checklists

Have you configured Azure Kubernetes Service (AKS) with cost optimization in mind?

- Use the Start and Stop feature in Azure Kubernetes Services (AKS).
- Enforce resource quotas at the namespace level.

Scalability checklist

- Assign pod requests and limits on AKS cluster.
- Define pod distribution budgets (PDB) for workloads.
- Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.
- Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.
- Use the Horizontal pod autoscaler to adjust the number of pods in a deployment depending on CPU utilization or other select metrics.
- Separate workloads into different node pools and consider scaling user node pools to zero.

AKS configuration recommendation

Explore the following recommendation to optimize your AKS configuration for service cost:

AKS RECOMMENDATION	DESCRIPTION
Use the Start and Stop feature in Azure Kubernetes Services (AKS).	The AKS Stop and Start cluster feature allows AKS customers to completely pause an AKS cluster, saving time and cost. The stop and start feature keeps cluster configurations in place and customers can pick up where they left off without reconfiguring the clusters.
Enforce resource quotas at the namespace level.	Resource quotas provide a way to reserve and limit resources across a development team or project. These quotas are defined on a namespace and can be used to set quotas on Compute resources, Storage resources, and Object counts. When you define resource quotas, all pods created in the namespace must provide limits or requests in their pod specifications. If they don't provide these values, you can reject the deployment.

Scalability recommendations

The following table reflects scalability recommendations and descriptions related to the overall AKS configuration recommendations:

SCALABILITY RECOMMENDATIONS	DESCRIPTION
Assign pod requests and limits on AKS cluster.	In your pod specifications, it's best practice to define requests and limits for CPU and memory consumption. If you don't include these values, the Kubernetes scheduler can't take into account the resources your applications require to aid in scheduling decisions. Set pod requests and limits on all pods in your YAML manifests. If the AKS cluster uses resource quotas, your deployment may be rejected if you don't define these values.
Define pod distribution budgets (PDB) for workloads.	To maintain the availability of applications, define PDBs to make sure that a minimum number of pods are available in the cluster.
Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.	The ability to automatically scale up or down the number of nodes in your AKS cluster lets you run an efficient, cost-effective cluster.
Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.	For example, workloads such as batch processing jobs, development, and testing environments, and large compute workloads may be good candidates for you to schedule on a spot node pool. Using spot VMs for nodes with your AKS cluster allows you to take advantage of unused capacity in Azure at a significant cost savings.
Separate workloads into different node pools and consider scaling user node pools to zero.	Unlike System node pools that always require running nodes, user node pools allow you to scale to `0`.

Next step

AKS and operational excellence

Azure Kubernetes Service (AKS) and operational excellence

11/19/2021 • 7 minutes to read • Edit Online

Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance.

To explore how AKS can bolster the operational excellence of your application workload, reference architecture guidance and best practices on the AKS Solution architectures page.

The following sections include configuration checklists, recommended configuration options, and supporting source artifacts specific to AKS.

Checklists

Have you configured Azure Kubernetes Service (AKS) with operational excellence in mind?

- Review AKS best practices documentation.
- Run multiple workloads in a single AKS cluster.
- Reboot nodes when updates and patches require it.
- Don't modify resources in the node resource group (for example MC_). You should *only* make modifications at cluster creation time, or with assistance from Azure Support.
- Use a central monitoring tool, such as Azure Monitor and App Insights to centrally collect metrics, logs, and diagnostics for troubleshooting purposes.
- Enable and review Kubernetes master node logs.
- Configure scraping of Prometheus metrics with Azure Monitor for containers.
- Use the AKS Uptime SLA for production grade clusters.
- Store container images within Azure Container Registry and enable geo-replication to replicate container images across leveraged AKS regions.
- Enable Azure Defender for container registries to enable vulnerability scanning for container images.
- Authenticate with Azure AD to Azure Container Registry.
- Use Availability Zones to maximize resilience within an Azure region by distributing AKS agent nodes across physically separate data centers.
- Use a template-based deployment using Terraform, Ansible, and others only. Make sure that all deployments are repeatable and traceable, and stored in a source code repo. Can be combined with GitOps.
- Adopt a multiregion strategy by deploying AKS clusters deployed across different Azure regions to maximize availability and provide business continuity.

Node pool design checklist

- Use Virtual Machine Scale Set (VMSS) VM set type for AKS node pools.
- Keep the System node pool isolated from application workloads.
- Use dedicated node pools for infrastructure tools that require high resource utilization, such as Istio, or have a special scale, or load behavior.
- Separate applications to dedicated node pools based on specific requirements.
- Use taints and tolerations to provide dedicated nodes and limit resource intensive applications.
- Consider the use of Virtual Nodes (vKubelet) with ACI for rapid, massive, and infinite scale.

Scalability checklist

- Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.
- Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.
- Use the Horizontal pod autoscaler to adjust the number of pods in a deployment depending on CPU utilization or other select metrics.
- Separate workloads into different node pools and consider scaling user node pools to zero.

AKS roadmap and GitHub release notes checklist

- Subscribe to the public AKS Roadmap Release Notes on GitHub to stay up to date on upcoming changes, improvements, Kubernetes version releases, and the deprecation of old releases.
- Regularly upgrade to a supported version of Kubernetes.
- Regularly process node image updates.
- Leverage AKS Cluster auto-upgrade with Planned Maintenance.

Security guidelines checklist

- Use Managed Identities to avoid managing and rotating service principles.
- Use AAD integration to take advantage of centralized account management, passwords, application access management, and identity protection.
- Use Kubernetes RBAC with AAD for least privilege and minimize granting administrator privileges to protect configuration, and secrets access.
- Limit access to Kubernetes cluster configuration file with Azure role-based access control.
- Limit access to actions that containers can perform. Provide the least number of permissions, and avoid the use of root, or privileged escalation.
- Evaluate the use of the built-in AppArmor security module to limit actions that containers can perform such as read, write, or execute, or system functions such as mounting files systems.
- Evaluate the use of the seccomp (secure computing). Seccomp works at the process level and allows you to limit the process calls that containers can perform.
- Use Pod Identities and Secrets Store CSI Driver with Azure Key Vault to protect secrets, certificates, and connection strings.
- Use Azure Security Center to provide AKS recommendations.
- Secure clusters and pods with Azure Policy.

AKS configuration recommendations

Explore the following table of recommendations to optimize your AKS configuration for operational excellence:

AKS RECOMMENDATION	DESCRIPTION
Review AKS best practices documentation.	To build and run applications successfully in AKS, there are some key considerations to understand and implement. These areas include multi-tenancy and scheduler features, cluster, and pod security, or business continuity and disaster recovery.
Configure scraping of Prometheus metrics with Azure Monitor for containers.	Azure Monitor for containers provides a seamless onboarding experience to collect Prometheus metrics. Reference Configure scraping of Prometheus metrics with Azure Monitor for containers for more information.

AKS RECOMMENDATION	DESCRIPTION
Authenticate with Azure AD to Azure Container Registry.	AKS and Azure AD enables authentication with Azure Container Registry without the use of K8s and `imagePullSecrets` secrets. Reference Authenticate with Azure Container Registry from Azure Kubernetes Service for more information.
Adopt a multiregion strategy by deploying AKS clusters deployed across different Azure regions to maximize availability and provide business continuity.	Internet facing workloads should leverage Azure Front Door, Azure Traffic Manager, or a third-party CDN to route traffic globally across AKS clusters.

Node pool design recommendations

The following table reflects node pool design recommendations and descriptions related to the overall AKS configuration recommendations:

NODE POOL DESIGN RECOMMENDATIONS	DESCRIPTION
Keep the System node pool isolated from application workloads.	System node pools require a VM SKU of at least `2` `vCPUs` and `4GB` memory. Reference System and user node pools for detailed requirements.
Separate applications to dedicated node pools based on specific requirements.	Such as GPU, high memory VMs, scale-to-zero, Spot VMs, and so on. Avoid large numbers of node pools to reduce extra management overhead.

Scalability recommendations

The following table reflects scalability recommendations and descriptions related to the overall AKS configuration recommendations:

SCALABILITY RECOMMENDATIONS	DESCRIPTION
Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.	The ability to automatically scale up or down the number of nodes in your AKS cluster lets you run an efficient, cost-effective cluster.
Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.	For example, workloads such as batch processing jobs, development, and testing environments, and large compute workloads may be good candidates for you to schedule on a spot node pool. Using spot VMs for nodes with your AKS cluster allows you to take advantage of unused capacity in Azure at a significant cost savings.
Separate workloads into different node pools and consider scaling user node pools to zero.	Unlike System node pools that always require running nodes, user node pools allow you to scale to `0`.

AKS roadmap and GitHub release notes recommendations

The following table reflects AKS roadmap and GitHub release notes recommendations, and descriptions related to the overall AKS configuration recommendations:

ROADMAP AND RELEASE NOTES RECOMMENDATIONS	DESCRIPTION

ROADMAP AND RELEASE NOTES RECOMMENDATIONS	DESCRIPTION
Subscribe to the AKS Roadmap and Release Notes on GitHub	Make sure that you're subscribed to the public AKS Roadmap Release Notes on GitHub to stay up to date on upcoming changes, improvements, and most importantly Kubernetes version releases, and the deprecation of old releases.
Regularly upgrade to a supported version of Kubernetes	AKS supports three minor versions of Kubernetes. When a new minor patch version is introduced, the oldest minor version and patch releases supported are retired. Minor updates to Kubernetes happen on a periodic basis. It's important to have a governance process to check and upgrade as needed to not fall out of support. For more information, reference Supported Kubernetes versions AKS.
Regularly process node image updates	AKS supports upgrading the images on a node to be up to date with the newest OS and runtime updates without updating the version of Kubernetes. The AKS team provides one new image version per week with the latest updates, including Linux or Windows patches.
Leverage AKS Cluster auto-upgrade with Planned Maintenance	AKS supports different auto-upgrade channels (08/18/21 in public preview) to upgrade AKS clusters to newer versions of Kubernetes and newer node images once available. Planned Maintenance (08/18/21 in public preview) can be used to define maintenance windows for these operations.

Security guideline recommendations

The following table reflects security guideline recommendations and descriptions related to the overall AKS configuration recommendations:

SECURITY GUIDELINE RECOMMENDATIONS	DESCRIPTION
Secure clusters and pods with Azure Policy	Azure Policy can help to apply at-scale enforcements and safeguards on your clusters in a centralized, consistent manner. It can also control what functions pods are granted and if anything is running against company policy. This access is defined through built-in policies provided by the Azure Policy Add-on for AKS. By providing more control over the security aspects of your pod's specification, like root privileges, Azure Policy enables stricter security adherence and visibility into what you've deployed in your cluster. If a pod doesn't meet conditions specified in the policy, Azure Policy can disallow the pod to start or flag a violation.

Source artifacts

To identify AKS clusters *not* using **RBAC**, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| where properties.enableRBAC == false
```

To identify AKS clusters *not* deployed using a **Managed Identity**, use the following query:

```
Resources
| where type =~ 'Microsoft.ContainerService/managedClusters'
| where isnull(identity)
```

Next step

AKS and performance efficiency

Azure Kubernetes Service (AKS) and performance efficiency

11/19/2021 • 2 minutes to read • Edit Online

Azure Kubernetes Service (AKS) simplifies deploying a managed Kubernetes cluster in Azure by offloading the operational overhead to Azure. As a hosted Kubernetes service, Azure handles critical tasks, like health monitoring and maintenance.

To explore how AKS can bolster the performance efficiency of your application workload, reference architecture guidance and best practices on the AKS Solution architectures page.

The following sections include a scalability checklist and recommended configuration options specific to AKS.

Scalability checklist

Have you configured Azure Kubernetes Service (AKS) with performance efficiency in mind?

- Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.
- Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.
- Use the Horizontal pod autoscaler to adjust the number of pods in a deployment depending on CPU utilization or other select metrics.
- Separate workloads into different node pools and consider scaling user node pools to zero.

Scalability recommendations

The following table reflects scalability recommendations and descriptions related to AKS configuration:

SCALABILITY RECOMMENDATIONS	DESCRIPTION
Enable cluster autoscaler to automatically adjust the number of agent nodes in response to resource constraints.	The ability to automatically scale up or down the number of nodes in your AKS cluster lets you run an efficient, cost-effective cluster.
Consider using Azure Spot VMs for workloads that can handle interruptions, early terminations, or evictions.	For example, workloads such as batch processing jobs, development, testing environments, and large compute workloads may be good candidates for you to schedule on a spot node pool. Using spot VMs for nodes with your AKS cluster allows you to take advantage of unused capacity in Azure at a significant cost savings.
Separate workloads into different node pools and consider scaling user node pools to zero.	Unlike System node pools that always require running nodes, user node pools allow you to scale to `0`.

Next step

Azure Functions and security

Azure Functions and security

11/19/2021 • 2 minutes to read • Edit Online

Azure Functions is a cloud service available on-demand that provides all the continually updated infrastructure and resources needed to run your applications. Functions allow you to write less code, maintain less infrastructure, and save on costs. Instead of worrying about deploying and maintaining servers, the cloud infrastructure provides all the up-to-date resources needed to keep your applications securely running.

For more information related to network security, reference Securing Azure Functions.

The following sections include a design consideration checklist and recommendations specific to Azure Functions, and security.

Design consideration checklist

Have you designed your workload and configured Azure Functions with security in mind?

- Evaluate if Azure Functions requires HTTP trigger.
- Treat Azure Functions code just like any other code.
- Use guidance available on Securing Azure Functions.
- Consider using Azure Functions Proxy to act as a facade.

Design consideration recommendations

The following table reflects design consideration recommendations and descriptions related to Azure Functions:

AZURE FUNCTIONS DESIGN RECOMMENDATIONS	DESCRIPTION
Evaluate if Azure Functions requires HTTP trigger.	Azure Functions supports multiple specific triggers and bindings. These include blob storage, Cosmos DB, Service Bus, and many more. If HTTP trigger is needed, then consider protecting that HTTP endpoint like any other web application. Common protection measures include keeping HTTP endpoint internal to specific Azure Virtual Networks by using Private endpoint connections or service endpoints. Consider using guidance available on Azure Functions networking options for more information. If Functions HTTP endpoint will be exposed to internet, then it's recommended to secure the endpoint behind a Web Application Firewall (WAF).
Treat Azure Functions code just like any other code.	Subject Azure Functions code to code scanning tools that are integrated with CI/CD pipeline.
Use guidance available on Securing Azure Functions.	This guidance addresses key security concerns such as operations, deployment, and network security.
Consider using Azure Functions Proxy to act as a facade.	Functions Proxy can inspect and modify incoming requests and responses.

Next step

Azure Service Fabric and reliability

11/19/2021 • 3 minutes to read • Edit Online

Azure Service Fabric is a distributed systems platform that makes it easy to:

- Package
- Deploy
- Manage scalable and reliable microservices, and containers.

For more information about how Azure Service Fabric creates a highly available and reliable state, reference the Reliability subsystem included in the Service Fabric architecture.

The following sections cover design considerations and configuration recommendations, specific to Azure Service Fabric, and reliability.

Design considerations

Azure Service Fabric doesn't provide its own SLA. The availability of Service Fabric clusters is based on the underlying Virtual Machine and storage resources used.

Virtual Machine Scale Sets also don't have an SLA, since the SLA for Virtual Machines applies here. If the Virtual Machine Scale Set includes Virtual Machines in at least two fault domains, the availability of the underlying Virtual Machines SLA for two or more instances applies. If the scale set contains a single Virtual Machine, the availability for a Single Instance Virtual Machine applies.

For more SLA information, reference Service Fabric SLA and Virtual Machine Scale Set SLA.

Checklist

Have you configured Azure Service Fabric with resiliency in mind?

- When using secrets such as connection strings and passwords in Service Fabric services, either retrieve secrets directly from Key Vault at runtime or use the Service Fabric Secrets Store.
- Use durability level Silver (5 VMs) or greater for production scenarios.
- For critical workloads, consider using Availability Zones for your Service Fabric clusters.
- To expose services on the Service Fabric cluster, use a reverse proxy such as the Service Fabric reverse proxy or Traefik. When exposing APIs hosted on the cluster, consider using Azure API Management.
- For production scenarios, use the Standard tier load balancer. The Basic SKU is free, but doesn't have an SLA.
- Apply Network Security Groups (NSG) to restrict traffic flow between subnets and node types. Ensure that the correct ports are opened for managing the cluster.
- Review the Service Fabric production readiness checklist.
- When using the Service Fabric Secret Store to distribute secrets, use a separate data encipherment certificate to encrypt the values.
- Don't use self-signed certificates for production scenarios. Either provision a certificate through your PKI or use a public certificate authority.
- Deploy certificates by adding them to Azure Key Vault and referencing the URI in your deployment.
- Create a process for monitoring the expiration date of certificates.
- Enable Azure Active Directory integration for your cluster to ensure users can access Service Fabric Explorer using their AAD credentials. Don't distribute the cluster certificate among users to access Explorer.

- Exclude the Service Fabric processes from Windows Defender to improve performance.
- Keep the different node types and gateway services on different subnets.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure Service Fabric configuration for service reliability:

AZURE SERVICE FABRIC RECOMMENDATION	DESCRIPTION
Use durability level Silver (5 VMs) or greater for production scenarios.	This level ensures the Azure infrastructure communicates with the Service Fabric controller on scheduling reboots, and so on.
Consider using Availability Zones for your Service Fabric clusters.	Deploy a primary NodeType (and by extension a virtual machine scale set) to each AZ, which ensures the Service Fabric system services are spread across zones.
Consider using Azure API Management to expose APIs hosted on the cluster.	API Management can integrate with Service Fabric directly.
Apply Network Security Groups (NSG) to restrict traffic flow between subnets and node types.	For example, you may have an API Management instance (one subnet), a frontend subnet (exposing a website directly), and a backend subnet (accessible only to frontend), each implemented on a different virtual machine scale set.
Use a separate data encipherment certificate to encrypt the values, when using the Service Fabric Secret Store to distribute secrets.	This certificate is deployed to the virtual machine scale set nodes to decrypt the secret values. When using this approach, ensure that secrets are inserted and encrypted at release time. Using this approach means that changing the secrets requires a deployment. Make sure your key-rotation process is fully automated to follow this process without downtime.
Create a process for monitoring the expiration date of certificates.	For example, Key Vault offers a feature that sends an email when `x%` of the certificate's lifespan has elapsed.
Exclude the Service Fabric processes from Windows Defender to improve performance.	By default, Windows Defender antivirus is installed on Windows Server 2016 and 2019. To reduce any performance impact and resource consumption overhead incurred by Windows Defender, and if your security policies allow you to exclude processes and paths for open-source software, you can exclude the Service Fabric executables from Defender scans.

Next step

Azure Service Fabric and security

Azure Service Fabric and security

11/19/2021 • 2 minutes to read • Edit Online

Azure Service Fabric is a distributed systems platform that makes it easy to:

- Package
- Deploy
- Manage scalable and reliable microservices, and containers.

For more information about how Azure Service Fabric creates a secure and reliable state, reference Azure Service Fabric security.

The following section covers configuration recommendations, specific to Azure Service Fabric, and security.

Checklist

Have you configured Azure Service Fabric with security in mind?

- Apply Network Security Groups (NSG) to restrict traffic flow between subnets and node types. Ensure that the correct ports are opened for managing the cluster.

Configuration recommendation

Consider the following recommendation to optimize your Azure Service Fabric configuration for security:

AZURE SERVICE FABRIC RECOMMENDATION	DESCRIPTION
Apply Network Security Groups (NSG) to restrict traffic flow between subnets and node types.	For example, you may have an API Management instance (one subnet), a frontend subnet (exposing a website directly), and a backend subnet (accessible only to frontend), each implemented on a different virtual machine scale set.

Next step

Azure Service Fabric and operational excellence

Azure Service Fabric and operational excellence

11/19/2021 • 2 minutes to read • Edit Online

Azure Service Fabric is a distributed systems platform that makes it easy to:

- Package
- Deploy
- Manage scalable and reliable microservices, and containers.

For more information about how Azure Service Fabric creates scalable and reliable production-ready workloads, reference the Production readiness checklist.

The following section covers configuration recommendations, specific to Azure Service Fabric, and operational excellence.

Checklist

Have you configured Azure Service Fabric with operational excellence in mind?

- When using secrets such as connection strings and passwords in Service Fabric services, either retrieve secrets directly from Key Vault at runtime or use the Service Fabric Secrets Store.
- Use durability level Silver (5 VMs) or greater for production scenarios.
- For critical workloads, consider using Availability Zones for your Service Fabric clusters.
- For production scenarios, use the Standard tier load balancer. The Basic SKU is free, but doesn't have an SLA.
- Apply Network Security Groups (NSG) to restrict traffic flow between subnets and node types. Ensure that the correct ports are opened for managing the cluster.
- Review the Service Fabric production readiness checklist.
- When using the Service Fabric Secret Store to distribute secrets, use a separate data encipherment certificate to encrypt the values.
- Don't use self-signed certificates for production scenarios. Either provision a certificate through your PKI or use a public certificate authority.
- Deploy certificates by adding them to Azure Key Vault and referencing the URI in your deployment.
- Create a process for monitoring the expiration date of certificates.
- Enable Azure Active Directory integration for your cluster to ensure users can access Service Fabric Explorer using their AAD credentials. Don't distribute the cluster certificate among users to access Explorer.
- Exclude the Service Fabric processes from Windows Defender to improve performance.
- Keep the different node types and gateway services on different subnets.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure Service Fabric configuration for operational excellence:

AZURE SERVICE FABRIC RECOMMENDATION	DESCRIPTION
Use durability level Silver (5 VMs) or greater for production scenarios.	This level ensures the Azure infrastructure communicates with the Service Fabric controller on scheduling reboots, and so on.

AZURE SERVICE FABRIC RECOMMENDATION	DESCRIPTION
Consider using Availability Zones for your Service Fabric clusters.	Deploy a primary NodeType (and by extension a virtual machine scale set) to each AZ, which ensures the Service Fabric system services are spread across zones.
Apply Network Security Groups (NSG) to restrict traffic flow between subnets and node types.	For example, you may have an API Management instance (one subnet), a frontend subnet (exposing a website directly), and a backend subnet (accessible only to frontend), each implemented on a different virtual machine scale set.
Use a separate data encipherment certificate to encrypt the values, when using the Service Fabric Secret Store to distribute secrets.	This certificate is deployed to the virtual machine scale set nodes to decrypt the secret values. When using this approach, ensure that secrets are inserted and encrypted at release time. Using this approach means that changing the secrets requires a deployment. Make sure your key-rotation process is fully automated to follow this process without downtime.
Create a process for monitoring the expiration date of certificates.	For example, Key Vault offers a feature that sends an email when `x%` of the certificate's lifespan has elapsed.
Exclude the Service Fabric processes from Windows Defender to improve performance.	By default, Windows Defender antivirus is installed on Windows Server 2016 and 2019. To reduce any performance impact and resource consumption overhead incurred by Windows Defender, and if your security policies allow you to exclude processes and paths for open-source software, you can [exclude](/azure/service-fabric/## Next step

Azure Service Fabric and performance efficiency

Azure Service Fabric and performance efficiency

11/19/2021 • 2 minutes to read • Edit Online

Azure Service Fabric is a distributed systems platform that makes it easy to:

- Package
- Deploy
- Manage scalable and reliable microservices, and containers.

For more information about how Azure Service Fabric can reduce performance issues for your workload with Service Fabric performance counters, reference Monitoring and diagnostic best practices for Azure Service Fabric.

The following section covers configuration recommendations, specific to Azure Service Fabric, and performance efficiency.

Checklist

Have you configured Azure Service Fabric with performance efficiency in mind?

- Exclude the Service Fabric processes from Windows Defender to improve performance.

Configuration recommendation

Consider the following recommendation to optimize your Azure Service Fabric configuration for performance efficiency:

AZURE SERVICE FABRIC RECOMMENDATION	DESCRIPTION
Exclude the Service Fabric processes from Windows Defender to improve performance.	By default, Windows Defender antivirus is installed on Windows Server 2016 and 2019. To reduce any performance impact and resource consumption overhead incurred by Windows Defender, and if your security policies allow you to exclude processes and paths for open-source software, you can exclude the Service Fabric executables from Defender scans.

Next step

Azure Virtual Machines and reliability

Virtual Machines and reliability

11/19/2021 • 6 minutes to read • Edit Online

Virtual Machines is an on-demand, scalable computing resource that gives you the flexibility of virtualization without having to buy and maintain physical hardware to run it.

For more information about how virtual machines support workload reliability, reference Availability options for Azure Virtual Machines.

The following sections cover design considerations and configuration recommendations specific to Virtual Machines and reliability.

Design considerations

Microsoft provides the following SLAs for virtual machines:

- `95%` SLA for single instance virtual machines using Standard HDD-Managed Disks for OS and Data disks.
- `99.5%` SLA for single instance virtual machines using Standard SSD-Managed Disks for OS and Data disks.
- `99.9%` SLA for single instance virtual machines using Premium SSD or Ultra Disk for all OS and Data disks.
- `99.95%` SLA for all virtual machines that have two or more instances in the same Availability Set or Dedicated Host Group.
- `99.99%` SLA for all virtual machines that have two or more instances deployed across two or more Availability Zones in the same region.

Checklist

Have you configured Virtual Machines with resiliency in mind?

- Azure Metadata Service Scheduled Events should be used to proactively respond to maintenance events, such as reboots, and limit disruption to virtual machines.
- Establish virtual machine Resource Health alerts to notify key stakeholders when resource health events occur.
- To ensure application scalability while navigating within disk sizing thresholds, it's highly recommended that applications be installed on data disks rather than on the OS disk. Managed Disks should be used for all virtual machine OS and Data disks to ensure resiliency across underlying storage stamps within a data center.
- Singleton workloads should use Premium Managed Disks to enhance resiliency and obtain a `99.9%` SLA as well as dedicated performance characteristics.
- Non-Singleton workloads should consider two or more replica instances with Managed disks (Standard or Premium) that are deployed within an Availability Set to obtain a `99.95%` SLA or across Availability Zones to obtain a `99.99%` SLA.
- Where appropriate, virtual machines should be deployed across Availability Zones to maximize resilience within a specific Azure region.
- Consider using proximity placement groups (PPGs) with Availability Zones (AZ) to have redundant in-zone VMs.
- Enable diagnostic logging for all virtual machines to ensure you route health metrics, boot diagnostics, and infrastructure logs to Log Analytics or an alternative log aggregation technology.
- Azure Backup should be used to back-up virtual machines in a Recovery Services Vault, to protect against

accidental data loss.

- Enable Soft Delete for the Recovery Services vault to protect against accidental or malicious deletion of backup data, ensuring the ability to recover.

Configuration recommendations

Explore the following table of recommendations to optimize your Virtual Machine configuration for service reliability:

VIRTUAL MACHINE RECOMMENDATION	DESCRIPTION
Azure Metadata Service Scheduled Events should be used to proactively respond to maintenance events, such as reboots, and limit disruption to virtual machines.	Scheduled Events is an Azure Metadata Service that gives your application time to prepare for virtual machine (VM) maintenance. It provides information about upcoming maintenance events (for example, reboot) so that your application can prepare for them and limit disruption.
Establish virtual machine Resource Health alerts to notify key stakeholders when resource health events occur.	An appropriate threshold for resource unavailability must be set to minimize signal to noise ratios so that transient faults don't generate an alert. For example, configuring a virtual machine alert with an unavailability threshold of one minute before an alert is triggered. Reference Resource Health Alerts for more information.
Where appropriate, virtual machines should be deployed across Availability Zones to maximize resilience within a specific Azure region.	Availability Zones offer unique physical locations within an Azure region, where each zone is made up of one or more datacenters equipped with independent power, cooling, and networking. Reference Datacenter Fault Tolerance and High availability and disaster recovery for IaaS apps for more information.
Consider using proximity placement groups (PPGs) with Availability Zones (AZ) to have redundant in-zone VMs.	It's not possible to create an Availability Set (AS) inside an Availability Zone (AZ) and it's also not possible to control the distribution of VMs within a single availability zone across different fault domains (FD), and update domains (UD). All VMs within a single availability zone might share a common power source and network switch, and can all be rebooted, or affected, by an outage or maintenance task at the same time. If you create VMs across different AZs, your VMs are effectively distributed across different FDs and UDs. If you want to achieve redundant in-zone VMs and cross-zone VMs, you should place the in-zone VMs in proximity placement groups within availability sets to ensure they won't all be rebooted at once. Go to Combine ASs and AZs with PPGs for detailed instructions.
Enable diagnostic logging for all virtual machines to ensure you route health metrics, boot diagnostics, and infrastructure logs to Log Analytics or an alternative log aggregation technology.	Platform logs provide detailed diagnostic and auditing information for Azure resources, and the Azure platform they depend on. Reference Overview of Azure platform logs for more information.
Enable Soft Delete for the Recovery Services vault to protect against accidental or malicious deletion of backup data, ensuring the ability to recover.	With Azure Backup Soft Delete, even if a malicious actor deletes a backup (or backup data is accidentally deleted), the backup data is kept for `14` more days, allowing the recovery of that backup item with no data loss. The extra `14` days of retention for backup data in the soft delete state don't incur any cost to you.

Supporting source artifacts

Use the following query to *identify standalone single instance VMs that aren't protected by a minimum SLA of at least* `99.5%`. The query will return all VM instances that aren't deployed within an Availability Set, across Availability Zones, and aren't using either Standard SSD or Premium SSD for both OS and Data disks. This query can be altered easily to identify all single instance VMs, including those using Premium Storage, which are protected by a minimum SLA of at least `99.5%`. Remove the trailing `where` condition:

```
Resources
| where
    type =~ 'Microsoft.Compute/virtualMachines'
        and isnull(properties.availabilitySet.id)
    or type =~ 'Microsoft.Compute/virtualMachineScaleSets'
        and sku.capacity <= 1
        or properties.platformFaultDomainCount <= 1
| where
    tags != '{"Skip":""}'
| where
    isnull(zones)
| where
    properties.storageProfile.osDisk.managedDisk.storageAccountType !in ('Premium_LRS'
    or properties.storageProfile.dataDisks.managedDisk.storageAccountType != 'Premium_LRS'
        and array_length(properties.storageProfile.dataDisks) != 0
```

The following query expands on the identification of standalone instances by *identifying any Availability Sets containing single instance VMs*, which are exposed to the same risks as standalone single instances outside of an Availability Set:

```
Resources
| where
    type =~ 'Microsoft.Compute/availabilitySets'
| where
    tags != '{"Skip":""}'
| where
    array_length(properties.virtualMachines) <= 1
| where
    properties.platformFaultDomainCount <= 1
```

Policy definitions

- Azure policy definition is to *audit standalone single instance VMs that aren't protected by an SLA*. It will flag an audit event for all Virtual Machine instances that aren't deployed within an Availability Set, across Availability Zones, and aren't using Premium Storage for both OS and Data disks. It also encompasses both Virtual Machine and Virtual Machine Scale Set resources.
- To identify resiliency risks to existing compute resources and support continuous compliance for new resources within a customer tenant, it's recommended you use Azure Policy and Azure Resource Graph to Audit the use of non-resilient deployment configurations.
- Azure policy definition is to audit Availability Sets containing single instance VMs that aren't protected by an SLA. It will flag an audit event for all Availability Sets that don't contain multiple instances.

Next step

Virtual machines and cost optimization

Virtual Machines and cost optimization

11/19/2021 • 2 minutes to read • Edit Online

Virtual Machines are an on-demand, scalable computing resource that gives you the flexibility of virtualization without having to buy and maintain physical hardware to run it.

For more information about how virtual machines support cost optimization for your workloads, reference Availability options for Azure Virtual Machines.

The following sections include a configuration checklist and recommendations specific to Virtual Machines and cost optimization.

Checklist

Have you configured Virtual Machines with cost optimization in mind?

- Shut down VM instances which aren't in use.
- Use Spot VMs when appropriate.
- Consider using Burstable (B) series VM sizes for VMs that are idle most of the time and have high usage for a certain period of time.
- Use Zone to Zone disaster recovery for virtual machines.
- Review SKUs that could benefit from Reserved Instances for one year, three years, or more.

Configuration recommendations

Explore the following table of recommendations to optimize your Virtual Machine configuration for service cost:

VIRTUAL MACHINE RECOMMENDATION	DESCRIPTION
Shut down VM instances which aren't in use.	Use the Start and Stop VMs during off-hours feature of virtual machines to minimize waste. Many configuration options exist to schedule start and stop times. The feature is suitable as a low-cost automation option. Azure Advisor evaluates virtual machines based on CPU and network usage over a time period. Advisor recommends actions like shut down or resize instances.
Use Spot VMs when appropriate.	Spot VMs are ideal for workloads that can be interrupted, such as highly parallel batch processing jobs. These VMs take advantage of the surplus capacity in Azure at a lower cost. They're also well suited for experimenting, developing, and testing large-scale solutions.
Consider using Burstable (B) series VM sizes for VMs that are idle most of the time and have high usage for a certain period of time.	The B-series VMs are ideal for workloads that don't need the full performance of the CPU continuously such as web servers, proof of concepts, small databases, and development build environments.

VIRTUAL MACHINE RECOMMENDATION	DESCRIPTION
Use Zone to Zone disaster recovery for virtual machines.	Replicate, failover, and failback your business-critical virtual machines within the same region with zones. Ideal for those customers that have complicated networking infrastructure and want to avoid the cost, and complexity of recreating infrastructure in a secondary region. For more information about regions, reference Products available by region.
Review SKUs that could benefit from Reserved Instances for one year, three years, or more.	Purchasing reserved instances is a way to reduce Azure costs for workloads with stable usage. Make sure you manage usage. If usage is too low, then you're paying for resources that aren't used. Keep RI instances simple and keep management overhead low to prevent increasing cost.

Next step

Virtual machines and operational excellence

Virtual Machines and operational excellence

11/19/2021 • 6 minutes to read • Edit Online

Virtual Machines is an on-demand, scalable computing resource that gives you the flexibility of virtualization without having to buy and maintain physical hardware to run it.

For more information about how virtual machines support operational excellence in your workload, reference Backup and restore options for virtual machines.

The following sections cover design considerations and configuration recommendations specific to Virtual Machines and operational excellence.

Design considerations

Microsoft provides the following SLAs for virtual machines:

- `95%` SLA for single instance virtual machines using Standard HDD-Managed Disks for OS and Data disks.
- `99.5%` SLA for single instance virtual machines using Standard SSD-Managed Disks for OS and Data disks.
- `99.9%` SLA for single instance virtual machines using Premium SSD or Ultra Disk for all OS and Data disks.
- `99.95%` SLA for all virtual machines that have two or more instances in the same Availability Set or Dedicated Host Group.
- `99.99%` SLA for all virtual machines that have two or more instances deployed across two or more Availability Zones in the same region.

Checklist

Have you configured Virtual Machines with operational excellence in mind?

- Azure Metadata Service Scheduled Events should be used to proactively respond to maintenance events, such as reboots, and limit disruption to virtual machines.
- Establish virtual machine Resource Health alerts to notify key stakeholders when resource health events occur.
- To ensure application scalability while navigating within disk sizing thresholds, it's highly recommended that applications be installed on data disks rather than on the OS disk. Managed disks should be used for all virtual machine OS and Data disks to ensure resiliency across underlying storage stamps within a data center.
- Singleton workloads should use Premium managed disks to enhance resiliency and obtain a `99.9%` SLA as well as dedicated performance characteristics.
- Non-Singleton workloads should consider two or more replica instances with managed disks (Standard or Premium) that are deployed within an Availability Set to obtain a `99.95%` SLA or across Availability Zones to obtain a `99.99%` SLA.
- Where appropriate, virtual machines should be deployed across Availability Zones to maximize resilience within a specific Azure region.
- Consider using proximity placement groups (PPGs) with Availability Zones (AZ) to have redundant in-zone VMs.
- Enable diagnostic logging for all virtual machines to ensure you route health metrics, boot diagnostics, and infrastructure logs to Log Analytics or an alternative log aggregation technology.
- Azure Backup should be used to back-up virtual machines in a Recovery Services Vault, to protect against

accidental data loss.

- Enable Soft Delete for the Recovery Services vault to protect against accidental or malicious deletion of backup data, ensuring the ability to recover.

Configuration recommendations

Explore the following table of recommendations to optimize your Virtual Machine configuration for service reliability:

VIRTUAL MACHINE RECOMMENDATION	DESCRIPTION
Azure Metadata Service Scheduled Events should be used to proactively respond to maintenance events, such as reboots, and limit disruption to virtual machines.	Scheduled Events is an Azure Metadata Service that gives your application time to prepare for virtual machine (VM) maintenance. It provides information about upcoming maintenance events (for example, reboot) so that your application can prepare for them and limit disruption.
Establish virtual machine Resource Health alerts to notify key stakeholders when resource health events occur.	An appropriate threshold for resource unavailability must be set to minimize signal to noise ratios so that transient faults don't generate an alert. For example, configuring a virtual machine alert with an unavailability threshold of one minute before an alert is triggered. Reference Resource Health Alerts for more information.
Where appropriate, virtual machines should be deployed across Availability Zones to maximize resilience within a specific Azure region.	Availability Zones offer unique physical locations within an Azure region, where each zone is made up of one or more datacenters equipped with independent power, cooling, and networking. Reference Datacenter Fault Tolerance and High availability and disaster recovery for IaaS apps for more information.
Consider using proximity placement groups (PPGs) with Availability Zones (AZ) to have redundant in-zone VMs.	It's not possible to create an Availability Set (AS) inside an Availability Zone (AZ) and it's also not possible to control the distribution of VMs within a single availability zone across different fault domains (FD), and update domains (UD). All VMs within a single availability zone might share a common power source and network switch, and can all be rebooted, or affected, by an outage or maintenance task at the same time. If you create VMs across different AZs, your VMs are effectively distributed across different FDs and UDs. If you want to achieve redundant in-zone VMs and cross-zone VMs, you should place the in-zone VMs in proximity placement groups within availability sets to ensure they won't all be rebooted at once. Go to Combine ASs and AZs with PPGs for detailed instructions.
Enable diagnostic logging for all virtual machines to ensure you route health metrics, boot diagnostics, and infrastructure logs to Log Analytics or an alternative log aggregation technology.	Platform logs provide detailed diagnostic and auditing information for Azure resources, and the Azure platform they depend on. Reference Overview of Azure platform logs for more information.
Enable Soft Delete for the Recovery Services vault to protect against accidental or malicious deletion of backup data, ensuring the ability to recover.	With Azure Backup Soft Delete, even if a malicious actor deletes a backup (or backup data is accidentally deleted), the backup data is kept for `14` more days, allowing the recovery of that backup item with no data loss. The extra `14` days of retention for backup data in the soft delete state don't incur any cost to you.

Supporting source artifacts

Use the following query to *identify standalone single instance VMs that aren't protected by a minimum SLA of at least* `99.5%`. The query will return all VM instances that aren't deployed within an Availability Set, across Availability Zones, and aren't using either Standard SSD or Premium SSD for both OS and Data disks. This query can be altered easily to identify all single instance VMs, including those using Premium Storage, which are protected by a minimum SLA of at least `99.5%`. Remove the trailing `where` condition:

```
Resources
| where
    type =~ 'Microsoft.Compute/virtualMachines'
        and isnull(properties.availabilitySet.id)
    or type =~ 'Microsoft.Compute/virtualMachineScaleSets'
        and sku.capacity <= 1
        or properties.platformFaultDomainCount <= 1
| where
    tags != '{"Skip":""}'
| where
    isnull(zones)
| where
    properties.storageProfile.osDisk.managedDisk.storageAccountType !in ('Premium_LRS'
    or properties.storageProfile.dataDisks.managedDisk.storageAccountType != 'Premium_LRS'
        and array_length(properties.storageProfile.dataDisks) != 0
```

The following query expands on the identification of standalone instances by *identifying any Availability Sets containing single instance VMs*, which are exposed to the same risks as standalone single instances outside of an Availability Set:

```
Resources
| where
    type =~ 'Microsoft.Compute/availabilitySets'
| where
    tags != '{"Skip":""}'
| where
    array_length(properties.virtualMachines) <= 1
| where
    properties.platformFaultDomainCount <= 1
```

Policy definitions

- Azure policy definition is to *audit standalone single instance VMs that aren't protected by an SLA*. It will flag an audit event for all Virtual Machine instances that aren't deployed within an Availability Set, across Availability Zones, and aren't using Premium Storage for both OS and Data disks. It also encompasses both Virtual Machine and Virtual Machine Scale Set resources.
- To identify resiliency risks to existing compute resources and support continuous compliance for new resources within a customer tenant, it's recommended you use Azure Policy and Azure Resource Graph to Audit the use of non-resilient deployment configurations.
- Azure policy definition is to audit Availability Sets containing single instance VMs that aren't protected by an SLA. It will flag an audit event for all Availability Sets that don't contain multiple instances.

Next step

Azure Cache for Redis and reliability

Azure Cache for Redis and reliability

11/19/2021 • 4 minutes to read • Edit Online

Azure Cache for Redis provides an in-memory data store based on the Redis (Remote Dictionary Server) software. It's a secure data cache and messaging broker that provides high throughput and low-latency access to data for applications.

Key concepts and best practices that support reliability include:

- High availability
- Failover and patching
- Connection resilience

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure Cache for Redis.

Design considerations

The Azure Cache for Redis Service Level Agreements (SLA) covers only Standard and Premium tier caches. Basic tier isn't covered.

Redis is an in-memory cache for key value pairs and has High Availability (HA), by default, except for Basic tier. There are three tiers for Azure Cache for Redis:

- *Basic*: *Not recommended for production workloads*. Basic tier is ideal for:
  - Single node
  - Multiple sizes
  - Development
  - Test
  - Non-critical workloads
- *Standard*: A replicated cache in a two-node primary and secondary configuration managed by Microsoft, with a high availability SLA.
- *Premium*: Includes all standard-tier features and includes the following other features:
  - Faster hardware and performance compared to Basic or Standard tier.
  - Larger cache size, up to `120GB`.
  - Data persistence, which includes Redis Database File (RDB) and Append Only File (AOF).
  - VNET support.
  - Clustering
  - Geo-Replication: A secondary cache is in another region and replicates data from the primary for disaster recovery. To failover to the secondary, the caches need to be unlinked manually and then the secondary is available for writes. The application writing to Redis needs to be updated with the secondary's cache connection string.
  - Availability Zones: Deploy the cache and replicas across availability zones.

> **NOTE**
> By default, each deployment will have one replica per shard. Persistence, clustering, and geo-replication are all disabled at this time with deployments that have more than one replica. Your nodes will be distributed evenly across all zones. You should have a replica count `>=` number of zones.

- Import and export.

Microsoft guarantees at least `99.9%` of the time that customers will have connectivity between the Cache Endpoints and Microsoft's Internet gateway.

Checklist

Have you configured Azure Cache for Redis with resiliency in mind?

- Schedule updates.
- Monitor the cache and set alerts.
- Deploy the cache within a VNET.
- Evaluate a partitioning strategy within Redis cache.
- Configure **Data Persistence** to save a copy of the cache to Azure Storage or use Geo-Replication, depending on the business requirement.
- Implement retry policies in the context of your Azure Redis Cache.
- Use one static or singleton implementation of the connection multiplexer to Redis and follow the best practices guide.
- Review How to administer Azure Cache for Redis.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure Cache for Redis configuration for service reliability:

RECOMMENDATION	DESCRIPTION
Schedule updates.	Schedule the days and times that Redis Server updates will be applied to the cache, which doesn't include Azure updates, or updates to the VM operating system.
Monitor the cache and set alerts.	Set alerts for exceptions, high CPU, high memory usage, server load, and evicted keys for insights about when to scale the cache. If the cache needs to be scaled, understanding when to scale is important because it will increase CPU during the scaling event to migrate data.
Deploy the cache within a VNET.	Gives the customer more control over the traffic that can connect to the cache. Make sure that the subnet has sufficient address space available to deploy the cache nodes and shards (cluster).

RECOMMENDATION	DESCRIPTION
Evaluate a partitioning strategy within Redis cache.	Partitioning a Redis data store involves splitting the data across instances of the Redis server. Each instance makes up a single partition. Azure Redis Cache abstracts the Redis services behind a facade and doesn't expose them directly. The simplest way to implement partitioning is to create multiple Azure Redis Cache instances and spread the data across them. You can associate each data item with an identifier (a partition key) that specifies which cache stores the data item. The client application logic can then use this identifier to route requests to the appropriate partition. This scheme is simple, but if the partitioning scheme changes (for example, if extra Azure Redis Cache instances are created), client applications may need to be reconfigured.
Configure **Data Persistence** to save a copy of the cache to Azure Storage or use Geo-Replication, depending on the business requirement.	*Data Persistence*: if the master and replica reboot, the data will be loaded automatically from the storage account. *Geo-Replication*: The secondary cache needs to be unlinked from the primary. The secondary will now become the primary and can receive *writes*.
Implement retry policies in the context of your Azure Redis Cache.	Most Azure services and client SDKs include a retry mechanism. These mechanisms differ because each service has different characteristics and requirements. Each retry mechanism is tuned to a specific service.
Review How to administer Azure Cache for Redis.	Understand how data loss can occur with cache reboots and how to test the application for resiliency.

Source artifacts

To identify Redis instances that aren't on the Premium tier, use the following query:

```
Resources
| where type == 'microsoft.cache/redis'
| where properties.sku.name != 'Premium'
```

Next step

Azure Cache for Redis and operational excellence

Azure Cache for Redis and operational excellence

11/19/2021 • 4 minutes to read • Edit Online

Azure Cache for Redis provides an in-memory data store based on the Redis (Remote Dictionary Server) software. It's a secure data cache and messaging broker that provides high throughput and low-latency access to data for applications.

Best practices that support operational excellence include:

- Server load management
- Memory management
- Performance testing

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure Cache for Redis.

Design considerations

The Azure Cache for Redis Service Level Agreements (SLA) covers only Standard and Premium tier caches. Basic tier isn't covered.

Redis is an in-memory cache for key value pairs and has High Availability (HA), by default, except for Basic tier. There are three tiers for Azure Cache for Redis:

- *Basic*: *Not recommended for production workloads*. Basic tier is ideal for:
  - Single node
  - Multiple sizes
  - Development
  - Test
  - Non-critical workloads
- *Standard*: A replicated cache in a two-node primary and secondary configuration managed by Microsoft, with a high availability SLA.
- *Premium*: Includes all standard-tier features and includes the following other features:
  - Faster hardware and performance compared to Basic or Standard tier.
  - Larger cache size, up to `120GB`.
  - Data persistence, which includes Redis Database File (RDB) and Append Only File (AOF).
  - VNET support.
  - Clustering
  - Geo-Replication: A secondary cache is in another region and replicates data from the primary for disaster recovery. To failover to the secondary, the caches need to be unlinked manually and then the secondary is available for writes. The application writing to Redis will need to be updated with the secondary's cache connection string.
  - Availability Zones: Deploy the cache and replicas across availability zones.

> **NOTE**
> By default, each deployment will have one replica per shard. Persistence, clustering, and geo-replication are all disabled at this time with deployments that have more than one replica. Your nodes will be distributed evenly across all zones. You should have a replica count `>=` number of zones.

- Import and export.

Microsoft guarantees at least `99.9%` of the time that customers will have connectivity between the Cache Endpoints and Microsoft's Internet gateway.

Checklist

Have you configured Azure Cache for Redis with operational excellence in mind?

- Schedule updates.
- Monitor the cache and set alerts.
- Deploy the cache within a VNET.
- Use the correct caching type (local, in role, managed, redis) within your solution.
- Configure **Data Persistence** to save a copy of the cache to Azure Storage or use Geo-Replication, depending on the business requirement.
- Use one static or singleton implementation of the connection multiplexer to Redis and follow the best practices guide.
- Review How to administer Azure Cache for Redis.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure Cache for Redis configuration for operational excellence:

RECOMMENDATION	DESCRIPTION
Schedule updates.	Schedule the days and times that Redis Server updates will be applied to the cache, which doesn't include Azure updates, or updates to the VM operating system.
Monitor the cache and set alerts.	Set alerts for exceptions, high CPU, high memory usage, server load, and evicted keys for insights about when to scale the cache. If the cache needs to be scaled, understanding when to scale is important because it will increase CPU during the scaling event to migrate data.
Deploy the cache within a VNET.	Gives the customer more control over the traffic that can connect to the cache. Make sure that the subnet has sufficient address space available to deploy the cache nodes and shards (cluster).

RECOMMENDATION	DESCRIPTION
Use the correct caching type (local, in role, managed, redis) within your solution.	Distributed applications typically implement either or both of the following strategies when caching data: - Using a private cache, where data is held locally on the machine that's running an instance of an application or service. - Using a shared cache, serving as a common source that can be accessed by multiple processes and machines. In both cases, caching can be performed client-side and server-side. Client-side caching is done by the process that provides the user interface for a system, such as a web browser or desktop application. Server-side caching is done by the process that provides the business services that are running remotely.
Configure **Data Persistence** to save a copy of the cache to Azure Storage or use Geo-Replication, depending on the business requirement.	*Data Persistence*: If the master and replica reboot, the data will be loaded automatically from the storage account. *Geo-Replication*: The secondary cache needs to be unlinked from the primary. The secondary will now become the primary and can receive *writes*.
Review How to administer Azure Cache for Redis.	Understand how data loss can occur with cache reboots and how to test the application for resiliency.

Source artifacts

To identify Redis instances that aren't on the Premium tier, use the following query:

```
Resources
| where type == 'microsoft.cache/redis'
| where properties.sku.name != 'Premium'
```

Next step

Azure Databricks and security

Azure Databricks and security

11/19/2021 • 2 minutes to read • Edit Online

Azure Databricks is a data analytics platform optimized for Azure cloud services. It offers three environments for developing data intensive applications:

- Databricks SQL
- Databricks Data Science and Engineering
- Databricks Machine Learning

To learn more about how Azure Databricks improves the security of big data analytics, reference Azure Databricks concepts.

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure Databricks.

Design considerations

All users' notebooks and notebook results are encrypted at rest, by default. If other requirements are in place, consider using customer-managed keys for notebooks.

Checklist

Have you configured Azure Databricks with security in mind?

- Use Azure Active Directory credential passthrough to avoid the need for service principals when communicating with Azure Data Lake Storage.
- Isolate your workspaces, compute, and data from public access. Make sure that only the right people have access and only through secure channels.
- Ensure that the cloud workspaces for your analytics are only accessible by properly managed users.
- Implement Azure Private Link.
- Restrict and monitor your virtual machines.
- Use Dynamic IP access lists to allow admins to access workspaces only from their corporate networks.
- Use the VNet injection functionality to enable more secure scenarios.
- Use diagnostic logs to audit workspace access and permissions.
- Consider using the Secure cluster connectivity feature and hub/spoke architecture to prevent opening ports, and assigning public IP addresses on cluster nodes.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure Databricks configuration for security:

RECOMMENDATION	DESCRIPTION
Ensure that the cloud workspaces for your analytics are only accessible by properly managed users.	Azure Active Directory can handle single sign-on for remote access. For extra security, reference Conditional Access.

RECOMMENDATION	DESCRIPTION
Implement Azure Private Link.	Ensure all traffic between users of your platform, the notebooks, and the compute clusters that process queries are encrypted and transmitted over the cloud provider's network backbone, inaccessible to the outside world.
Restrict and monitor your virtual machines.	Clusters, which execute queries, should have SSH and network access restricted to prevent installation of arbitrary packages. Clusters should use only images that are periodically scanned for vulnerabilities.
Use the VNet injection functionality to enable more secure scenarios.	Such as: - Connecting to other Azure services using service endpoints. - Connecting to on-premises data sources, taking advantage of user-defined routes. - Connecting to a network virtual appliance to inspect all outbound traffic and take actions according to allow and deny rules. - Using custom DNS. - Deploying Azure Databricks clusters in existing virtual networks.
Use diagnostic logs to audit workspace access and permissions.	Use audit logs to see privileged activity in a workspace, cluster resizing, files, and folders shared on the cluster.

Source artifacts

Azure Databricks source artifacts include the Databricks blog: Best practices to secure an enterprise-scale data platform.

Next step

Azure Database for MySQL and cost optimization

Azure Database for MySQL and cost optimization

11/19/2021 • 2 minutes to read • Edit Online

Azure Database for MySQL is a relational database service in the Microsoft cloud based on the MySQL Community Edition. You can use either Single Server or Flexible Server to host a MySQL database in Azure. It's a fully managed database as a service offering that can handle mission-critical workloads with predictable performance and dynamic scalability.

For more information about how Azure Database for MySQL supports cost optimization for your workload, reference Server concepts, specifically, Stop/Start an Azure Database for MySQL.

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure Database for MySQL.

Design considerations

Azure Database for MySQL includes the following design considerations:

- Take advantage of the scaling capabilities of Azure Database for MySQL to lower consumption cost whenever possible. To scale your database up and down, as needed, reference the following Microsoft Support article, which covers the automation process using runbooks: How to autoscale an Azure Database for MySQL/PostgreSQL instance with Azure run books and Python.
- Plan your Recovery Point Objective (RPO) according to your operation level requirement. There's no extra charge for backup storage for up to `100%` of your total provisioned server storage. Extra consumption of backup storage will be charged in `GB/month`.
- The cloud native design of the Single-Server service allows it to support `99.99%` of availability, eliminating the cost of passive *hot* standby.
- Consider using Flexible Server SKU for non-production workloads. Flexible servers provide better cost optimization controls with ability to stop and start your server. They provide a burstable compute tier that is ideal for workloads that don't need continuous full compute capacity.

Checklist

Have you configured Azure Database for MySQL with cost optimization in mind?

- Choose the appropriate server size for your workload.
- Consider Reserved Capacity for Azure Database for MySQL Single Server.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure Database for MySQL configuration for cost optimization:

RECOMMENDATION	DESCRIPTION
Choose the appropriate server size for your workload.	Configuration options: Single Server and Flexible Server.

RECOMMENDATION	DESCRIPTION
Consider Reserved Capacity for Azure Database for MySQL Single Server.	Compute costs associated with Azure Database For MySQL Single Server Reservation Discount. Once you've determined the total compute capacity and performance tier for Azure Database for MySQL in a region, this information can be used to reserve the capacity. The reservation can span one or three years. You can realize significant cost optimization with this commitment.

Azure Database for PostgreSQL and cost optimization

Azure Database for PostgreSQL and cost optimization

11/19/2021 • 2 minutes to read • Edit Online

Azure Database for PostgreSQL is a relational database service in the Microsoft cloud based on the PostgreSQL Community Edition.

You can choose between three deployment modes, such as Single Server, Flexible Server, and Hyperscale (Citus):

- *Single Server*: A fully managed database service with minimal requirements for database customizations. This platform is designed to handle most database management functions, such as:
  - Patching
  - Backups
  - High Availability
  - Security with minimal user configuration and control

    This service offers three pricing tiers and allows you to pay only for the resources you need, and only when you need them.
- *Flexible Server*: A fully managed database service designed to provide more granular control and flexibility over database management functions and configuration settings based on user requirements. This service provides better cost optimization controls because you can stop and start the server, and the burstable compute tier.
- *Hyperscale*: This option uses sharding to horizontally scale across multiple machines. Hyperscale is best for applications that require greater scale where workloads approach `100GB` of data.

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure Database for PostgreSQL.

Design considerations

Azure Database for PostgreSQL includes the following design considerations:

- Hyperscale (Citus) provides dynamic scalability without the cost of manual sharding with low application rearchitecture required.

  Distributing table rows across multiple PostgreSQL servers is a key technique for scalable queries in Hyperscale (Citus). Together, multiple nodes can hold more data than a traditional database, and in many cases can use worker CPUs in parallel to execute queries potentially lowering the database costs. Follow this Shard data on worker nodes tutorial to practice this potential savings architecture pattern.
- Consider using Flexible Server SKU for non-production workloads.

  Flexible servers provide better cost optimization controls with ability to stop and start your server, and burstable compute tier that is ideal for workloads that don't need continuous full compute capacity.
- Plan your Recovery Point Objective (RPO) according to your operation level requirement.

  There's no extra charge for backup storage for up to `100%` of your total provisioned server storage. Extra

consumption of backup storage will be charged in GB/month.

- Take advantage of the scaling capabilities of Azure Database for PostgreSQL to lower consumption cost whenever possible.

  This Microsoft Support article about How to autoscale an Azure Database for MySQL/PostgreSQL instance with Azure runbooks and Python covers the automation process using runbooks to scale your database up and down, as needed.

- The cloud native design of the Single- Server service allows it to support `99.99%` of availability eliminating the cost of passive *hot* standby.

Checklist

Have you configured Azure Database for PostgreSQL with cost optimization in mind?

- Choose the appropriate server size for your workload.
- Consider Reserved Capacity for Azure Database for PostgreSQL Single Server and Hyperscale (Citus).

Configuration recommendations

Explore the following table of recommendations to optimize your Azure Database for PostgreSQL configuration for cost optimization:

RECOMMENDATION	DESCRIPTION
Choose the appropriate server size for your workload.	Configuration options: Single Server, Flexible Server, Hyperscale (Citus).
Consider Reserved Capacity for Azure Database for PostgreSQL Single Server and Hyperscale (Citus).	Compute costs associated with Azure Database For PostgreSQL Single Server Reservation Discount and Hyperscale (Citus) Reservation Discount. Once the total compute capacity and performance tier for Azure Database for PostgreSQL in a region is determined, this information can be used to reserve the capacity. The reservation can span one or three years. You can realize significant cost optimization with this commitment.

Next step

Azure SQL Database and reliability

Azure SQL Database and reliability

11/19/2021 • 5 minutes to read • Edit Online

Azure SQL Database is a fully managed platform as a service (PaaS) database engine that handles most of the database management functions without user involvement. Management functions include:

- Upgrades
- Patches
- Backups
- Monitoring

This service allows you to create a highly available and high-performance data storage layer for your Azure applications and workloads. Azure SQL Database is always running on the latest stable version of the SQL Server database engine and patched OS with `99.99%` availability.

For more information about how Azure SQL Database promotes reliability and enables your business to continue operating during disruptions, reference Availability capabilities.

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure SQL Database and reliability.

Design considerations

Azure SQL Database includes the following design considerations:

- Azure SQL Database Business Critical tier configured with geo-replication has a guaranteed Recovery time objective (RTO) of `30` seconds for `100%` of deployed hours.
- Use *sharding* to distribute data and processes across many identically structured databases. Sharding provides an alternative to traditional scale-up approaches for cost and elasticity. Consider using sharding to partition the database horizontally. Sharding can provide fault isolation. For more information, reference Scaling out with Azure SQL Database.
- Azure SQL Database Business Critical or Premium tiers not configured for Zone Redundant Deployments, General Purpose, Standard, or Basic tiers, or Hyperscale tier with two or more replicas have an availability guarantee. For more information about the availability guarantee, reference SLA for Azure SQL Database.
- Provides built-in regional high availability and turnkey geo-replication to any Azure region. It includes intelligence to support self-driving features, such as:
  - Performance tuning
  - Threat monitoring
  - Vulnerability assessments
  - Fully automated patching and updating of the code base
- Define an application performance SLA and monitor it with alerts. Quickly detect when your application performance inadvertently degrades below an acceptable level, which is important to maintain high resiliency. Use the monitoring solution previously defined to set alerts on key query performance metrics so you can take action when the performance breaks the SLA. Go to Monitor Your Database for more information.
- Use geo-restore to recover from a service outage. You can restore a database on any SQL Database server or an instance database on any managed instance in any Azure region from the most recent geo-

replicated backups. Geo-restore uses a geo-replicated backup as its source. You can request geo-restore even if the database or datacenter is inaccessible because of an outage. Geo-restore restores a database from a geo-redundant backup. For more information, reference Recover an Azure SQL database using automated database backups.

- Use the Business Critical tier configured with geo-replication, which has a guaranteed Recovery point objective (RPO) of `5` seconds for `100%` of deployed hours.
- PaaS capabilities built into Azure SQL Database enable you to focus on the domain-specific database administration and optimization activities that are critical for your business.
- Use point-in-time restore to recover from human error. Point-in-time restore returns your database to an earlier point in time to recover data from changes done inadvertently. For more information, read the Point-in-time restore (PITR) documentation.
- Business Critical or Premium tiers are configured as Zone Redundant Deployments which have an availability guarantee. For more information about the availability guarantee, reference SLA for Azure SQL Database.

Checklist

Have you configured Azure SQL Database with reliability in mind?

- Use Active Geo-Replication to create a readable secondary in a different region.
- Use Auto Failover Groups that can include one or multiple databases, typically used by the same application.
- Use a Zone-Redundant database.
- Monitor your Azure SQL Database in near-real time to detect reliability incidents.
- Implement Retry Logic.
- Back up your keys.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure SQL Database configuration for reliability:

RECOMMENDATION	DESCRIPTION
Use Active Geo-Replication to create a readable secondary in a different region.	If your primary database fails, perform a manual failover to the secondary database. Until you fail over, the secondary database remains read-only. Active geo-replication enables you to create readable replicas and manually failover to any replica if there is a datacenter outage or application upgrade. Up to four secondaries are supported in the same or different regions, and the secondaries can also be used for read-only access queries. The failover must be initiated manually by the application or the user. After failover, the new primary has a different connection end point.
Use Auto Failover Groups that can include one or multiple databases, typically used by the same application.	You can use the readable secondary databases to offload read-only query workloads. Because autofailover groups involve multiple databases, these databases must be configured on the primary server. Autofailover groups support replication of all databases in the group to only one secondary server or instance in a different region. Learn more about AutoFailover Groups and DR design.

RECOMMENDATION	DESCRIPTION
Use a Zone-Redundant database.	By default, the cluster of nodes for the premium availability model is created in the same datacenter. With the introduction of Azure Availability Zones, SQL Database can place different replicas of the Business Critical database to different availability zones in the same region. To eliminate a single point of failure, the control ring is also duplicated across multiple zones as three gateway rings (GW). The routing to a specific gateway ring is controlled by Azure Traffic Manager (ATM). Because the zone redundant configuration in the Premium or Business Critical service tiers doesn't create extra database redundancy, you can enable it at no extra cost. Learn more about Zone-redundant databases.
Monitor your Azure SQL Database in near-real time to detect reliability incidents.	Use one of the available solutions to monitor SQL DB to detect potential reliability incidents early and make your databases more reliable. Choose a near real-time monitoring solution to quickly react to incidents. Reference Azure SQL Analytics for more information.
Implement Retry Logic.	Although Azure SQL Database is resilient when it concerns transitive infrastructure failures, these failures might affect your connectivity. When a transient error occurs while working with SQL Database, make sure your code can retry the call. For more information, reference how to implement retry logic.
Back up your keys.	If you're not using encryption keys in Azure Key Vault to protect your data, back up your keys.

Next step

Azure SQL Database and cost optimization

Azure SQL Database and cost optimization

11/19/2021 • 2 minutes to read • Edit Online

Azure SQL Database is a fully managed platform as a service (PaaS) database engine that handles most of the database management functions without user involvement. Management functions include:

- Upgrades
- Patches
- Backups
- Monitoring

This service allows you to create a highly available and high-performance data storage layer for your Azure applications and workloads. SQL Database includes built-in intelligence that helps you dramatically reduce the costs of running and managing databases through automatic performance monitoring and tuning.

For more information about how Azure SQL Database provides cost-saving features, reference Plan and manage costs for Azure SQL Database.

The following sections include a configuration checklist and recommended configuration options specific to Azure SQL Database and cost optimization.

Checklist

Have you configured Azure SQL Database with cost optimization in mind?

- Evaluate DTU usage.
- Evaluate Azure SQL Database Serverless.
- Optimize queries.
- Consider reserved capacity for Azure SQL Database.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure SQL Database configuration for cost savings:

RECOMMENDATION	DESCRIPTION
Evaluate DTU usage.	Evaluate the DTU usage for all Databases and determine if they've been sized and provisioned correctly. For non-prod Databases, consider using Basic Tier or S0 and configure the DTUs, as applicable. The DTUs can be scaled on demand, for example, when running a load test and more.

RECOMMENDATION	DESCRIPTION
Evaluate Azure SQL Database Serverless.	Consider using Azure SQL Database Serverless over Provisioned Computing Tier. Serverless is a compute tier for single databases that automatically scales compute based on workload demand and bills for the amount of compute used per second. The serverless compute tier also automatically pauses databases during inactive periods when only storage is billed. It automatically resumes databases when activity returns. Azure SQL Database serverless isn't suited for all scenarios. If you have a database that isn't always heavily used and if you have periods of complete inactivity, serverless is a solution that guarantees performance and saves costs.
Optimize queries.	Optimize the queries, tables, and databases using Query Performance Insights and Performance Recommendations to help reduce resource consumption, and arrive at appropriate configuration.
Consider reserved capacity for Azure SQL Database.	You can reduce compute costs associated with Azure SQL Database by using Reservation Discount. Once you've determined the total compute capacity and performance tier for Azure SQL databases in a region, you can use this information to reserve the capacity. The reservation can span one or three years. For more information, reference Save costs for resources with reserved capacity.

Next step

Azure SQL Database and operational excellence

Azure SQL Database and operational excellence

11/19/2021 • 5 minutes to read • Edit Online

Azure SQL Database is a fully managed platform as a service (PaaS) database engine that handles most of the database management functions without user involvement. Management functions include:

- Upgrades
- Patches
- Backups
- Monitoring

This service allows you to create a highly available and high-performance data storage layer for your Azure applications and workloads. Azure SQL Database provides advanced monitoring and tuning capabilities backed by artificial intelligence to help you troubleshoot and maximize the performance of your databases and solutions.

For more information about how Azure SQL Database promotes operational excellence and enables your business to continue operating during disruptions, reference Monitoring and performance tuning in Azure SQL Database.

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure SQL Database, and operational excellence.

Design considerations

Azure SQL Database includes the following design considerations:

- Azure SQL Database Business Critical tier configured with geo-replication has a guaranteed Recovery time objective (RTO) of `30` seconds for `100%` of deployed hours.
- Use *sharding* to distribute data and processes across many identically structured databases. Sharding provides an alternative to traditional scale-up approaches for cost and elasticity. Consider using sharding to partition the database horizontally. Sharding can provide fault isolation. For more information, reference Scaling out with Azure SQL Database.
- Azure SQL Database Business Critical or Premium tiers not configured for Zone Redundant Deployments, General Purpose, Standard, or Basic tiers, or Hyperscale tier with two or more replicas have an availability guarantee. For more information, reference SLA for Azure SQL Database.
- Provides built-in regional high availability and turnkey geo-replication to any Azure region. It includes intelligence to support self-driving features, such as:
  - Performance tuning
  - Threat monitoring
  - Vulnerability assessments
  - Fully automated patching and updating of the code base
- Define an application performance SLA and monitor it with alerts. Quickly detect when your application performance inadvertently degrades below an acceptable level, which is important to maintain high resiliency. Use the monitoring solution previously defined to set alerts on key query performance metrics so you can take action when the performance breaks the SLA. Go to Monitor Your Database for more information.

- Use geo-restore to recover from a service outage. You can restore a database on any SQL Database server or an instance database on any managed instance in any Azure region from the most recent geo-replicated backups. Geo-restore uses a geo-replicated backup as its source. You can request geo-restore even if the database or datacenter is inaccessible because of an outage. Geo-restore restores a database from a geo-redundant backup. For more information, reference Recover an Azure SQL database using automated database backups.
- Use the Business Critical tier configured with geo-replication, which has a guaranteed Recovery point objective (RPO) of `5` seconds for `100%` of deployed hours.
- PaaS capabilities built into Azure SQL Database enable you to focus on the domain-specific database administration and optimization activities that are critical for your business.
- Use point-in-time restore to recover from human error. Point-in-time restore returns your database to an earlier point in time to recover data from changes done inadvertently. For more information, read the Point-in-time restore (PITR) documentation.
- Business Critical or Premium tiers are configured as Zone Redundant Deployments. For more information about the availability guarantee, reference SLA for Azure SQL Database .

Checklist

Have you configured Azure SQL Database with operational excellence in mind?

- Use Active Geo-Replication to create a readable secondary in a different region.
- Use Auto Failover Groups that can include one or multiple databases, typically used by the same application.
- Use a Zone-Redundant database.
- Monitor your Azure SQL Database in near-real time to detect reliability incidents.
- Implement Retry Logic.
- Back up your keys.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure SQL Database configuration for operational excellence:

RECOMMENDATION	DESCRIPTION
Use Active Geo-Replication to create a readable secondary in a different region.	If your primary database fails, perform a manual failover to the secondary database. Until you fail over, the secondary database remains read-only. Active geo-replication enables you to create readable replicas and manually failover to any replica if there is a datacenter outage or application upgrade. Up to four secondaries are supported in the same or different regions, and the secondaries can also be used for read-only access queries. The failover must be initiated manually by the application or the user. After failover, the new primary has a different connection end point.
Use Auto Failover Groups that can include one or multiple databases, typically used by the same application.	You can use the readable secondary databases to offload read-only query workloads. Because autofailover groups involve multiple databases, these databases must be configured on the primary server. Autofailover groups support replication of all databases in the group to only one secondary server or instance in a different region. Learn more about Auto-Failover Groups and DR design.

RECOMMENDATION	DESCRIPTION
Use a Zone-Redundant database.	By default, the cluster of nodes for the premium availability model is created in the same datacenter. With the introduction of Azure Availability Zones, SQL Database can place different replicas of the Business Critical database to different availability zones in the same region. To eliminate a single point of failure, the control ring is also duplicated across multiple zones as three gateway rings (GW). The routing to a specific gateway ring is controlled by Azure Traffic Manager (ATM). Because the zone redundant configuration in the Premium or Business Critical service tiers doesn't create extra database redundancy, you can enable it at no extra cost. Learn more about Zone-redundant databases.
Monitor your Azure SQL Database in near-real time to detect reliability incidents.	Use one of the available solutions to monitor SQL DB to detect potential reliability incidents early and make your databases more reliable. Choose a near real-time monitoring solution to quickly react to incidents. Reference Azure SQL Analytics for more information.
Implement Retry Logic.	Although Azure SQL Database is resilient when it concerns transitive infrastructure failures, these failures might affect your connectivity. When a transient error occurs while working with SQL Database, make sure your code can retry the call. For more information, reference how to implement retry logic.
Back up your keys.	If you're not using encryption keys in Azure Key Vault to protect your data, back up your keys.

Next step

Azure SQL Managed Instance and reliability

Azure SQL Managed Instance and reliability

11/19/2021 • 3 minutes to read • Edit Online

Azure SQL Managed Instance is the intelligent, scalable cloud database service that combines the broadest SQL Server database engine compatibility with all the benefits of a fully managed and evergreen platform as a service.

The goal of the high availability architecture in SQL Managed Instance is to guarantee that your database is up and running without worrying about the impact of maintenance operations and outages. This solution is designed to:

- Ensure that committed data is never lost because of failures.
- Ensure that maintenance failures don't affect your workload.
- Ensure that the database won't be a single point of failure in your software architecture.

For more information about how Azure SQL Managed Instance supports application and workload resilience, reference the following articles:

- High availability for Azure SQL Managed Instance
- Use autofailover groups to enable transparent and coordinated geo-failover of multiple databases

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure SQL Managed Instance, and reliability.

Design considerations

Azure SQL Managed Instance includes the following design considerations:

- Define an application performance SLA and monitor it with alerts. Detecting quickly when your application performance inadvertently degrades below an acceptable level is important to maintain high resiliency. Use a monitoring solution to set alerts on key query performance metrics so you can take action when the performance breaks the SLA.
- Use point-in-time restore to recover from human error. Point-in-time restore returns your database to an earlier point in time to recover data from changes done inadvertently. For more information, read the Point-in-time-restore (PITR) documentation for managed instance.
- Use geo-restore to recover from a service outage. Geo-restore restores a database from a geo-redundant backup into a managed instance in a different region. For more information, reference Recover a database using Geo-restore documentation.
- Consider the time required for certain operations. Make sure you separate time to thoroughly test the amount of time required to scale up and down your existing managed instance, and to create a new managed instance. This timing practice ensures that you understand completely how time consuming operations will affect your RTO and RPO.

Checklist

Have you configured Azure SQL Managed Instance with reliability in mind?

- Use the Business Critical Tier.
- Configure a secondary instance and an Autofailover group to enable failover to another region.
- Implement Retry Logic.
- Monitor your SQL MI instance in near-real time to detect reliability incidents.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure SQL Managed Instance configuration for reliability:

RECOMMENDATION	DESCRIPTION
Use the Business Critical Tier.	This tier provides higher resiliency to failures and faster failover times because of the underlying HA architecture, among other benefits. For more information, reference SQL Managed Instance High availability.
Configure a secondary instance and an Autofailover group to enable failover to another region.	If an outage impacts one or more of the databases in the managed instance, you can manually or automatically failover all the databases inside the instance to a secondary region. For more information, read the Autofailover groups documentation for managed instance.
Implement Retry Logic.	Although Azure SQL MI is resilient to transitive infrastructure failures, these failures might affect your connectivity. When a transient error occurs while working with SQL MI, make sure your code can retry the call. For more information, reference how to implement retry logic.
Monitor your SQL MI instance in near-real time to detect reliability incidents.	Use one of the available solutions to monitor your SQL MI to detect potential reliability incidents early and make your databases more reliable. Choose a near real-time monitoring solution to quickly react to incidents. For more information, check out the Azure SQL Managed Instance monitoring options.

Next step

Azure SQL Managed Instance and operational excellence

Azure SQL Managed Instance and operational excellence

11/19/2021 • 3 minutes to read • Edit Online

Azure SQL Managed Instance is the intelligent, scalable cloud database service that combines the broadest SQL Server database engine compatibility with all the benefits of a fully managed and evergreen platform as a service.

The goal of the high availability architecture in SQL Managed Instance is to guarantee that your database is up and running without worrying about the impact of maintenance operations and outages. This solution is designed to:

- Ensure that committed data is never lost because of failures.
- Ensure that maintenance failures don't affect your workload.
- Ensure that the database won't be a single point of failure in your software architecture.

For more information about how Azure SQL Managed Instance supports operational excellence for your application workloads, reference the following articles:

- Overview of Azure SQL Managed Instance management operations
- Monitoring Azure SQL Managed Instance management operations

The following sections include design considerations, a configuration checklist, and recommended configuration options specific to Azure SQL Managed Instance, and operational excellence.

Design considerations

Azure SQL Managed Instance includes the following design considerations:

- Define an application performance SLA and monitor it with alerts. Detecting quickly when your application performance inadvertently degrades below an acceptable level is important to maintain high resiliency. Use a monitoring solution to set alerts on key query performance metrics so you can take action when the performance breaks the SLA.
- Use point-in-time restore to recover from human error. Point-in-time restore returns your database to an earlier point in time to recover data from changes done inadvertently. For more information, read the Point-in-time-restore (PITR) documentation for managed instance.
- Use geo-restore to recover from a service outage. Geo-restore restores a database from a geo-redundant backup into a managed instance in a different region. For more information, reference Recover a database using Geo-restore documentation.
- Consider the time required for certain operations. Make sure you separate time to thoroughly test the amount of time required to scale up and down your existing managed instance, and to create a new managed instance. This timing practice ensures that you understand completely how time consuming operations will affect your RTO and RPO.

Checklist

Have you configured Azure SQL Managed Instance with operational excellence in mind?

- Use the Business Critical Tier.
- Configure a secondary instance and an Autofailover group to enable failover to another region.

- Implement Retry Logic.
- Monitor your SQL MI instance in near-real time to detect reliability incidents.

Configuration recommendations

Explore the following table of recommendations to optimize your Azure SQL Managed Instance configuration for operational excellence:

RECOMMENDATION	DESCRIPTION
Use the Business Critical Tier.	This tier provides higher resiliency to failures and faster failover times because of the underlying HA architecture, among other benefits. For more information, reference SQL Managed Instance High availability.
Configure a secondary instance and an Autofailover group to enable failover to another region.	If an outage impacts one or more of the databases in the managed instance, you can manually or automatically failover all the databases inside the instance to a secondary region. For more information, read the Autofailover groups documentation for managed instance.
Implement Retry Logic.	Although Azure SQL MI is resilient to transitive infrastructure failures, these failures might affect your connectivity. When a transient error occurs while working with SQL MI, make sure your code can retry the call. For more information, reference how to implement retry logic.
Monitor your SQL MI instance in near-real time to detect reliability incidents.	Use one of the available solutions to monitor your SQL MI to detect potential reliability incidents early and make your databases more reliable. Choose a near real-time monitoring solution to quickly react to incidents. For more information, check out the Azure SQL Managed Instance monitoring options.

Next step

Cosmos DB and reliability

Cosmos DB and reliability

11/19/2021 • 8 minutes to read • Edit Online

Cosmos DB is a fully managed NoSQL database for modern app development.

Key features include:

- Guaranteed speed at any scale
- Simplified application development
- Mission-critical ready
- Fully managed and cost effective

To understand how Cosmos DB bolsters resiliency for your application workload, reference the following articles:

- Distribute your data globally with Azure Cosmos DB
- How does Azure Cosmos DB provide high availability
- Consistency levels in Azure Cosmos DB
- Configure Azure Cosmos DB account with periodic backup

The following sections include design considerations, a configuration checklist, recommended configuration options, and source artifacts specific to Cosmos DB and reliability.

Design considerations

Cosmos DB includes the following design considerations:

- SLA for read availability for Database Accounts spanning two or more Azure regions.
- SLAs for throughput, consistency, availability, and latency.
- SLA for both read and write availability with the configuration of multiple Azure regions as writable endpoints.

For more granular information about SLAs specific to this product, reference Cosmos DB Service Level Agreements.

Checklist

Have you configured Cosmos DB with reliability in mind?

- Deploy Cosmos DB and the application in the region that corresponds to end users.
- If multi-master option is enabled on Cosmos DB, it's important to understand Conflict Types and Resolution Policies.
- Start with, Session, the default consistency level.
- Change the consistency level, depending on the data operation and usage.
- Evaluate connectivity modes and connection protocols in Cosmos DB.
- Configure preferred locations.
- Specify index precisions.
- Use Azure Monitor to see the provisioned autoscale max `RU/s` (Autoscale Max Throughput) and the `RU/s` the system is currently scaled to (Provisioned Throughput).
- Understand your traffic pattern to pick the right option for provisioned throughput types.

- *New applications*: If you don't know your traffic pattern yet, start at the entry point `RU/s` to avoid over-provisioning in the beginning.
- *Existing applications*: Use Azure Monitor metrics to determine if your traffic pattern is suitable for autoscale.
- *Existing applications*: Find the normalized request unit consumption metric of your database or container.
- *Existing applications*: The closer the number is to `100%`, the more you're fully using your provisioned `RU/s`.
- Set provisioned `RU/s` to `T` for all hours in a month.
- Enable automatic failover when you configure Cosmos accounts used for production workloads.
- Implement retry logic in your client.
- For query-intensive workloads, use Windows `64-bit` instead of Linux or Windows `32-bit` host processing.
- To reduce latency and CPU jitter, enable accelerated networking on client virtual machines in both Windows and Linux.
- Increase the number of threads and tasks.
- To avoid network latency, collocate client in the same region as Cosmos DB.
- Call OpenAsync to avoid startup latency on first request.
- Scale out client applications across multiple servers if client consumes more than `50,000` `RU/s`.
- Select a partition key.
- Ensure the partition key is a property that has a value that doesn't change.
- You can't change a partition key after it's created with the collection.
- Ensure the partition key has a high cardinality.
- Ensure the partition key spreads RU consumption and data storage evenly across all logical partitions.
- Ensure you're running read queries with the partitioned column to reduce RU consumption and latency.
- Evaluate ways to improve data performance.
- Configure data replication to ensure Cosmos DB meets the SLAs.

Configuration recommendations

Explore the following table of recommendations to optimize your Cosmos DB configuration for reliability:

RECOMMENDATION	DESCRIPTION
Deploy Cosmos DB and the application in the region that corresponds to end users.	There are two common scenarios for configuring two or more regions: Delivering low-latency access to data to end users no matter where they're located around the globe. Adding regional resiliency for business continuity and disaster recovery (BCDR). For delivering low-latency to end users, it's recommended to deploy both the application and add Cosmos DB in the regions that correspond to where the application's users are located.
Start with, Session, the default consistency level.	It's the recommended consistency level to start with as it receives data later, but in the same order as the writes.
Change the consistency level, depending on the data operation and usage.	Depending on the type of data stored, different consistency levels may be changed on a per request basis. For example, if log data is written to Cosmos DB, an **Eventual** consistency may be relevant, but if writing e-commerce transactions, then **Strong** may be more appropriate.

RECOMMENDATION	DESCRIPTION
Evaluate connectivity modes and connection protocols in Cosmos DB.	Cosmos DB supports two connectivity modes. In *Gateway mode*, requests are always made to the Cosmos DB gateway, which forwards it to the corresponding data partitions. In *Direct connectivity mode*, the client fetches the mapping of tables to partitions, and requests are made directly against data partitions. We recommend Direct, the default mode. Cosmos DB supports two connection protocols: `HTTPS` and `TCP`, which is the default. `TCP` is recommended because it's more lightweight.
Configure preferred locations.	Setting preferred locations can improve query performance. To take advantage of global distribution, client applications can specify the ordered preference list of regions to be used to perform document operations, which can be done by setting the connection policy. Based on the Cosmos DB account configuration, current regional availability and the preference list specified, the most optimal endpoint will be chosen by the SQL SDK to perform write and read operations. This preference list is specified when initializing a connection using the SQL SDKs. The SDKs accept an optional parameter, `PreferredLocations`, that is an ordered list of Azure regions. The SDK will automatically send all writes to the current write region. All reads will be sent to the first available region in the `PreferredLocations` list. If the request fails, the client will fail down the list to the next region, and so on. The SDKs will only attempt to read from the regions specified in `PreferredLocations`.
Specify index precisions.	Setting these values appropriately can improve query performance and reduce throughput requests. You can use index precision to make trade-offs between index storage overhead and query performance. For numbers, we recommend using the default precision configuration of `-1` (maximum). Because numbers are `8` bytes in JSON, this is equivalent to a configuration of `8` bytes. Choosing a lower value for precision, such as `1` through `7`, means that values within some ranges map to the same index entry. You reduce index storage space, but query execution might have to process more documents. It consumes more throughput in request units. Index precision configuration has more practical application with string ranges. Because strings can be any arbitrary length, the choice of the index precision might affect the performance of string range queries. It also may affect the amount of index storage space that's required. String Range indexes can be configured with `1` through `100` or `-1` (maximum).
Existing applications: Find the normalized request unit consumption metric of your database or container.	*Normalized* usage is a measure of how much you're currently using your standard (manual) provisioned throughput.
Set provisioned `RU/s` to `T` for all hours in a month.	If you set provisioned `RU/s` to `T` and use the full amount for `66%` of the hours or more, it's estimated you'll save with standard (manual) provisioned `RU/s`. If you set autoscale max `RU/s` to `Tmax` and use the full amount `Tmax` for `66%` of the hours or less, it's estimated you'll save with autoscale.

RECOMMENDATION	DESCRIPTION
Scale out client applications across multiple servers if client consumes more than `50,000` `RU/s`.	There could be a bottleneck because of the machine capping out on CPU or network usage.
Ensure the partition key spreads RU consumption and data storage evenly across all logical partitions.	This spread ensures even RU consumption and storage distribution across your physical partitions.
Evaluate ways to improve data performance.	Best practices for query performance: - Connection policy: Use direct connection mode. - Connection Policy: Use the `TCP` protocol `Call OpenAsync` to avoid startup latency on first request. - Collocate clients in the same Azure region for performance. - Increase number of threads and tasks. - Install the most recent SDK. - Use a singleton Cosmos DB client for the lifetime of your application. - Increase `System.Net` MaxConnections per host when using Gateway mode. - Tune parallel queries for partitioned collections. - Turn on server-side GC. - Implement backoff at `RetryAfter` intervals. - Scale out your client-workload. - Cache document URIs for lower read latency. - Tune the page size for queries and read feeds for better performance. - Use 64-bit host processing. - Exclude unused paths from indexing for faster writes. - Measure and tune for lower request units and second usage. - Handle rate limiting and request rates that are too large. - Design for smaller documents for higher throughput.
Configure data replication to ensure Cosmos DB meets the SLAs	If you've replicated your data in more than one data center, Cosmos DB automatically rolls over your operations should a regional data center go offline. You can create a prioritized list of failover regions using the regions in which your data is replicated. Even within a single data center, Cosmos DB automatically replicates data for high availability giving you the choice of consistency levels.

Source artifacts

To check for `cosmosdb` instances where automatic failover isn't enabled, use the following query:

```
Resources
|where  type =~ 'Microsoft.DocumentDb/databaseAccounts'
|where properties.enableAutomaticFailover!=True
```

Use the following query to see the list of multiregion writes:

```
resources
| where type == "microsoft.documentdb/databaseaccounts"
 and properties.enableMultipleWriteLocations == "true"
```

To view consistency levels for your Cosmos DB accounts, use the following query:

```
Resources
| project name, type, location, consistencyLevel = properties.consistencyPolicy.defaultConsistencyLevel
| where type == "microsoft.documentdb/databaseaccounts"
| order by name asc
```

To check if multilocation isn't selected, use the following query:

```
Resources
|where  type =~ 'Microsoft.DocumentDb/databaseAccounts'
|where array_length( properties.locations) <=1
```

Learn more

- High Availability in Cosmos DB
- Autoscale FAQ
- Performance Tips for Cosmos DB

Next step

Cosmos DB and operational excellence

Cosmos DB and operational excellence

11/19/2021 • 4 minutes to read • Edit Online

Cosmos DB is a fully managed NoSQL database for modern app development.

Key features include:

- Guaranteed speed at any scale
- Simplified application development
- Mission-critical ready
- Fully managed and cost effective

To understand how Cosmos DB promotes operational excellence for your application workload, reference the following articles:

- Monitor Azure Cosmos DB
- Monitor and debug with insights in Azure Cosmos DB
- Visualize Azure Cosmos DB data by using the Power BI connector

The following sections include design considerations, a configuration checklist, recommended configuration options, and source artifacts specific to Cosmos DB, and operational excellence.

Design considerations

Cosmos DB includes the following design considerations:

- SLA for read availability for Database Accounts spanning two or more Azure regions.
- SLAs for throughput, consistency, availability, and latency.
- SLA for both read and write availability with the configuration of multiple Azure regions as writable endpoints.

For more granular information specific to this product, reference Cosmos DB Service Level Agreements.

Checklist

Have you configured Cosmos DB with operational excellence in mind?

- Monitor for normal and abnormal activity.
- If multi-master option is enabled on Cosmos DB, it's important to understand Conflict Types and Resolution Policies.
- Start with, Session, the default consistency level.
- Use Azure Monitor to see the provisioned autoscale max `RU/s` (Autoscale Max Throughput) and the `RU/s` the system is currently scaled to (Provisioned Throughput).
- Understand your traffic pattern to pick the right option for provisioned throughput types.
- *New applications*: If you don't know your traffic pattern yet, start at the entry point `RU/s` to avoid over-provisioning in the beginning.
- *Existing applications*: Use Azure Monitor metrics to determine if your traffic pattern is suitable for autoscale.
- *Existing applications*: Find the normalized request unit consumption metric of your database or container.
- *Existing applications*: The closer the number is to `100%`, the more you're fully using your provisioned `RU/s`.
- Set provisioned `RU/s` to `T` for all hours in a month.

- Enable automatic failover when you configure Cosmos accounts used for production workloads.
- Implement retry logic in your client.
- For query-intensive workloads, use Windows `64-bit` instead of Linux or Windows `32-bit` host processing.
- To reduce latency and CPU jitter, enable accelerated networking on client virtual machines in both Windows and Linux.
- Increase the number of threads and tasks.
- To avoid network latency, collocate client in the same region as Cosmos DB.
- Call OpenAsync to avoid startup latency on first request.
- Scale out client applications across multiple servers if client consumes more than `50,000` `RU/s`.
- Select a partition key.
- Ensure the partition key is a property that has a value that doesn't change.
- You can't change a partition key after it's created with the collection.
- Ensure the partition key has a high cardinality.
- Ensure the partition key spreads RU consumption and data storage evenly across all logical partitions.
- Ensure you're running read queries with the partitioned column to reduce RU consumption and latency.

Configuration recommendations

Explore the following table of recommendations to optimize operational excellence for your Cosmos DB configuration:

RECOMMENDATION	DESCRIPTION
Monitor for normal and abnormal activity.	The Azure Activity Log is a subscription log that provides insight into subscription-level events that have occurred in Azure. The Activity Log reports control plane events for your subscriptions under the Administrative category. Using the Activity Log, you can determine the *what*, *who*, and *when* for any write operations (`PUT`, `POST`, `DELETE`) taken on the resources in your subscription. You can also understand the status of the operation and other relevant properties. The Activity Log differs from Diagnostic Logs. Activity Logs provide data about the operations on a resource from the outside (the *control plane*). In the Azure Cosmos DB context, some of the control plane operations include create collection, list keys, delete keys, list database, and more. Diagnostic Logs are emitted by a resource and provide information about the operation of that resource (the *data plane*). Some of the data plane diagnostic log examples include delete, insert, ReadFeed operation, and more.
Start with, Session, the default consistency level.	It's the recommended consistency level to start with as it receives data later, but in the same order as the writes.
Existing applications: Find the normalized request unit consumption metric of your database or container.	*Normalized* usage is a measure of how much you're currently using your standard (manual) provisioned throughput.
Set provisioned `RU/s` to `T` for all hours in a month.	If you set provisioned `RU/s` to `T` and use the full amount for `66%` of the hours or more, it's estimated you'll save with standard (manual) provisioned `RU/s`. If you set autoscale max `RU/s` to `Tmax` and use the full amount `Tmax` for `66%` of the hours or less, it's estimated you'll save with autoscale.

RECOMMENDATION	DESCRIPTION
Scale out client applications across multiple servers if client consumes more than `50,000` `RU/s`.	There could be a bottleneck because of the machine capping out on CPU or network usage.
Ensure the partition key spreads RU consumption and data storage evenly across all logical partitions.	This spread ensures even RU consumption and storage distribution across your physical partitions.

Source artifacts

To check for `cosmosdb` instances where automatic failover isn't enabled, use the following query:

```
Resources
|where  type =~ 'Microsoft.DocumentDb/databaseAccounts'
|where properties.enableAutomaticFailover!=True
```

Use the following query to see the list of multiregion writes:

```
resources
| where type == "microsoft.documentdb/databaseaccounts"
 and properties.enableMultipleWriteLocations == "true"
```

To view consistency levels for your Cosmos DB accounts, use the following query:

```
Resources
| project name, type, location, consistencyLevel = properties.consistencyPolicy.defaultConsistencyLevel
| where type == "microsoft.documentdb/databaseaccounts"
| order by name asc
```

To check if multilocation isn't selected, use the following query:

```
Resources
|where  type =~ 'Microsoft.DocumentDb/databaseAccounts'
|where array_length( properties.locations) <=1
```

Learn more

- Autoscale FAQ
- Performance Tips for Cosmos DB

Next step

Azure Stack Hub and reliability

Solutions for the retail industry

11/19/2021 • 4 minutes to read • Edit Online

Retail is one of the fastest growing industries worldwide, generating some of the biggest revenues and accounting to almost a third of American jobs. The core of retail industry is selling products and services to consumers, through channels such as, storefront, catalog, television, and online. Retailers can enhance or reimagine their customer's journey using Microsoft Azure services by:

- keeping their supply chains agile and efficient,
- unlocking new opportunities with data and analytics,
- creating innovative customer experiences using mixed reality, AI, and IoT, and
- building a personalized and secure multi-channel retail experience for customers.

Using Azure services, retailers can easily achieve these goals. For use cases and customer stories, visit Azure for retail. Microsoft is also revolutionizing the retail industry, by providing a comprehensive retail package, Microsoft Cloud for Retail.

> **NOTE**
>
> Learn more about a retail company's journey to cloud adoption, in Cloud adoption for the retail industry.

Architecture guides for retail

The following articles provide more details about retail architectural topics. Although they are mostly conceptual, they can also include implementation details.

GUIDE	SUMMARY	TECHNOLOGY FOCUS
Data Management in Retail	Primer for how to ingest, prepare, store, analyze, and take action on data, for the retail industry.	Databases
Migrating your e-commerce solution to Azure	Learn how to move an existing e-commerce solution to the cloud. The three stages are to rehost, refactor, and rebuild your solution.	Migration
Optimize and reuse an existing recommendation system	The process of successfully reusing and improving an existing recommendation system that is written in R.	AI/ML
Visual search in Retail with CosmosDB	This document focuses on the AI concept of visual search and offers a few key considerations on its implementation. It provides a workflow example and maps its stages to the relevant Azure technologies.	Databases

GUIDE	SUMMARY	TECHNOLOGY FOCUS
SKU optimization for consumer brands	Topics include automating decision making, SKU assortment optimization, descriptive analytics, predictive analytics, parametric models, non-parametric models, implementation details, data output and reporting, and security considerations.	Analytics

Architectures for retail

The following articles provide detailed analysis of architectures developed and recommended for the retail industry.

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Batch scoring with R models to forecast sales	Perform batch scoring with R models using Azure Batch. Azure Batch works well with intrinsically parallel workloads and includes job scheduling and compute management.	AI/ML
Build a Real-time Recommendation API on Azure	Build a recommendation engine that can be generalized for products, movies, news, and other consumer services, using Azure Databricks, Azure Machine Learning, Azure Cosmos DB, and Azure Kubernetes Service.	AI/ML
Movie recommendations on Azure	Automate movie and product recommendations by using an Azure Data Science Virtual Machine to train an Azure Machine Learning model.	AI/ML
Scalable personalization on Azure	Build a *content-based* personalized recommendation system for customers by learning their preferences from past purchases and interactions.	AI/ML
Data warehousing and analytics	Build an insightful sales and marketing solution with a data pipeline that integrates large amounts of data from multiple sources into a unified analytics platform in Azure.	Analytics
Stream processing with Azure Databricks	Use Azure Databricks to build an end-to-end stream processing pipeline for a taxi company, to collect, and analyze trip and fare data from multiple devices.	Analytics
Stream processing with Azure Stream Analytics	Use Azure Stream Analytics to build an end-to-end stream processing pipeline for a taxi company, to collect, and analyze trip and fare data from multiple devices.	Analytics

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
IBM z/OS online transaction processing on Azure	With a dynamically adaptable infrastructure, businesses can realize and launch their products quickly to delight their users. Learn how to migrate a z/OS mainframe OLTP application to a secure, scalable, and highly available system in the cloud.	Mainframe
E-commerce front end	Implement a scalable and cost-effective e-commerce front end using Azure platform as a service (PaaS) tools.	Web
Intelligent product search engine for e-commerce	Use Azure Cognitive Search, a dedicated search service, to dramatically increase the relevance of search results for your e-commerce customers.	Web
Magento e-commerce platform in Azure Kubernetes Service	Learn how to deploy and host Magento, an open-source e-commerce platform, on Azure.	Web
Scalable order processing	Build a highly scalable and resilient architecture for online order processing, using managed Azure services, such as Cosmos DB and HDInsight.	Web
Retail - Buy online, pickup in store (BOPIS)	Develop an efficient and secure curbside pickup process on Azure.	Web

Solution ideas for retail

The following are other ideas that you can use as a starting point for your retail solution.

AI.

- Commerce Chatbot with Azure Bot Service
- FAQ Chatbot with data champion model
- Customer Feedback and Analytics
- Interactive Voice Response Bot
- Optimize Marketing with Machine Learning
- Personalized Offers
- Personalized marketing solutions
- Predictive Marketing with Machine Learning
- Product recommendations for retail
- Retail Assistant with Visual Capabilities

Analytics.

- Big data analytics with Azure Data Explorer
- Demand Forecasting with Azure Machine Learning
- Demand Forecasting and Price Optimization
- Demand Forecasting for Shipping and Distribution

- Interactive Price Analytics

Databases.

- Retail and e-commerce using Azure MySQL
- Retail and e-commerce using Azure PostgreSQL
- Retail and e-commerce using Cosmos DB

Mixed Reality.

- Facilities management powered by mixed reality and IoT

Networking.

- Video capture and analytics for retail

Web.

- E-commerce website running in secured App Service Environment
- Architect scalable e-commerce web app
- Scalable Episerver marketing website
- Scalable Sitecore marketing website
- Simple digital marketing website

Solutions for the finance industry

11/19/2021 • 2 minutes to read • Edit Online

The finance industry includes a broad spectrum of entities such as banks, investment companies, insurance companies, and real estate firms, engaged in the funding and money management for individuals, businesses, and governments. Besides data security concerns, financial institutions face unique issues such as, heavy reliance on traditional mainframe systems, cyber and technology risks, compliance issues, increasing competition, and customer expectations. By modernizing and digitally transforming financial systems to move to cloud platforms such as Microsoft Azure, financial institutes can mitigate these issues and provide more value to their customers.

With digital transformation, financial institutions can leverage the speed and security of the cloud and use its capabilities to offer differentiated customer experiences, manage risks, and fight fraud. To learn more, visit Azure for financial services. Banking and capital market institutions can drive innovative cloud solutions with Azure; learn from relevant use cases and documentation at Azure for banking and capital markets. Microsoft also provides a complete set of capabilities across various platforms in the form of Microsoft Cloud for Financial Services.

Architectures for finance

The following articles provide detailed analysis of architectures recommended for the finance industry.

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Decentralized trust between banks	Learn how to establish a trusted environment for information sharing without resorting to a centralized database, in banks or other financial institutions.	Blockchain
Replicate and sync mainframe data in Azure	Replicate and sync mainframe data to Azure for digital transformation of traditional banking systems.	Mainframe
Modernize mainframe & midrange data	End to end modernization plan for mainframe and midrange data sources.	Mainframe
Refactor IBM z/OS mainframe Coupling Facility (CF) to Azure	Learn how to leverage Azure services for scale-out performance and high availability, comparable to IBM z/OS mainframe systems with Coupling Facilities (CFs).	Mainframe
Banking system cloud transformation on Azure	Learn how a major bank modernized its financial transaction system while keeping compatibility with its existing payment system.	Migration

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Patterns and implementations in banking cloud transformation	Learn the design patterns and implementations used for the Banking system cloud transformation on Azure.	Migration
JMeter implementation reference for load testing pipeline solution	Learn about an implementation for a scalable cloud load testing pipeline used for the Banking system cloud transformation on Azure.	Migration
Real-time fraud detection	Learn how to analyze data in real time to detect fraudulent transactions or other anomalous activity.	Security

Solution ideas for finance

The following are some additional ideas that you can use as a starting point for your finance solution.

- Auditing, risk, and compliance management
- Business Process Management
- HPC System and Big Compute Solutions
- HPC Risk Analysis Template
- Loan ChargeOff Prediction with Azure HDInsight Spark Clusters
- Loan ChargeOff Prediction with SQL Server
- Loan Credit Risk + Default Modeling
- Loan Credit Risk with SQL Server
- Unlock Legacy Data with Azure Stack

Solutions for the healthcare industry

11/19/2021 • 2 minutes to read • Edit Online

The healthcare industry includes various systems that provide curative, preventative, rehabilitative, and palliative care to patients. Proper management of these systems enables healthcare providers and managers provide high-quality care and treatment for their patients. With Azure cloud and other Microsoft services, you can now create highly efficient and resilient healthcare systems that take care of not only the patient-provider interactions, but also provide clinical and data insights, leading to a more patient-centric strategy for the healthcare institute.

Modernization and digital transformation of healthcare facilities is all the more important during the current COVID-19 global pandemic.

Learn how you can use Microsoft Azure services to digitize, modernize, and enhance your healthcare solution at Azure for healthcare. Microsoft also provides a comprehensive platform for the healthcare industry, Microsoft Cloud for Healthcare, which includes components from Dynamics 365 and Microsoft 365, in addition to Azure.

Architectures for healthcare

The following articles provide detailed analysis of architectures developed and recommended for the healthcare industry.

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Virtual health on Microsoft Cloud for Healthcare	Use Microsoft Cloud for Healthcare, a software package created for the healthcare industry, to build an architecture for scheduling and following up on virtual visits between patients, providers, and care managers.	Web
Clinical insights with Microsoft Cloud for Healthcare	Use Microsoft Cloud for Healthcare to collect, analyze, and visualize medical and health insights, that can be used to improve healthcare operations.	Web
Consumer health portal on Azure	Learn how to develop a consumer health portal using Azure services, to track statistics from wearables, engage with medical providers, and monitor health habits, built on a foundation of the Azure Well Architected Framework.	Web
Confidential computing for healthcare	Use Azure confidential computing to encrypt medical and patient data, for secure collaboration between hospitals and third-party diagnostic providers.	Security

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Health Data Consortium on Azure	Use the Azure Data Platform, and Azure Data Share to create an environment where healthcare organizations can appropriately, and securely share data with partner organizations to support activities like clinical trials and research.	Data
Precision Medicine Pipeline with Genomics	Use Microsoft Genomics and the Azure Data Platform to perform analysis and reporting for scenarios like precision medicine and genetic profiling.	Data/Analytics
Predict Hospital Readmissions with Machine Learning	Predict the readmissions of diabetic patients using Azure Data, AI, and Analytics tools through the different personas of Data Professionals throughout the process.	Data/AI
Build a telehealth system with Azure	Explore a customer's implementation of a telehealth system using Azure cloud services.	Containers

Solution ideas for healthcare

The following are some additional ideas that you can use as a starting point for your healthcare solution.

- Population Health Management for Healthcare
- Medical Data Storage Solutions
- HIPAA and HITRUST compliant health data AI
- Remote Patient Monitoring Solutions
- Predict Length of Stay and Patient Flow
- Predict Length of Stay in Hospitals
- Contactless IoT interfaces with Azure intelligent edge
- COVID-19 Safe Solutions with IoT Edge
- IoT Connected Platform for COVID-19 detection and prevention
- UVEN smart and secure disinfection and lighting

Solutions for the government industry

11/19/2021 • 2 minutes to read • Edit Online

Microsoft Azure provides a mission-critical cloud platform, Azure Government, that delivers breakthrough innovation to US government customers and their partners. US federal, state, local, and tribal governments and their partners can have secure and dedicated access to this platform, with operations controlled by screened US citizens.

Using Azure Government, you can:

- test and deploy secure, highly-available, and performant mission-critical apps,
- create custom web experiences,
- gain insights from your data by using AI and analytics capabilities.

Azure Government offers a broad level of certifications to simplify critical government compliance requirements. To learn more about this government-focused cloud platform, visit Azure Government.

Microsoft is committed to provide government agencies with innovative technology solutions across health and human services, critical infrastructure, public safety & justice, and tax, finance, and revenue. Learn more at Cloud computing for government.

Architectures for government

The following articles provide detailed analysis of architectures developed and recommended for the government industry.

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Azure Automation in a hybrid environment	Extend automated management and configuration from Azure to on-premises and other cloud providers.	Hybrid/Multicloud
Azure Automation Update Management	Design a hybrid update management solution to manage updates on both Microsoft Azure and on-premises Windows and Linux computers.	Hybrid/Multicloud
Computer forensics Chain of Custody in Azure	Ensure a valid Chain of Custody (CoC) in acquiring, storing, and accessing of digital evidence to support criminal investigations or civil proceedings.	Management/Governance
Hybrid Security Monitoring using Microsoft Defender for Cloud and Microsoft Sentinel	Monitor the security configuration and telemetry of on-premises and Azure operating system workloads.	Hybrid/Multicloud

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Vision classifier model with Azure Custom Vision Cognitive Service	Combine AI and Internet of Things (IoT) by using Azure Custom Vision to classify images taken by a simulated drone.	AI/ML
Web app private connectivity to Azure SQL database	Set up private connectivity from an Azure Web App to Azure Platform-as-a-Service (PaaS) services.	Security
Azure Virtual Desktop for the enterprise	Use Azure Virtual Desktop to build virtualized desktop infrastructure (VDI) solutions at enterprise scale, covering 1,000 virtual desktops and above.	Hybrid/Multicloud

Solution ideas for government

The following article provides an idea that you can use as a starting point for your government solution.

- DevSecOps in GitHub

Solutions for the manufacturing industry

11/19/2021 • 2 minutes to read • Edit Online

Manufacturing sector, a hallmark of the modern industrialized world, encompasses all steps from procuring raw materials to transforming into final product. Starting from household manufacturing in the pre-industrial era, this sector has evolved through stages such as mechanized assembly lines and automation, every new development adding to faster and more efficient manufacturing processes. Cloud computing can bring forth the next revolution for manufacturing companies by transforming their IT infrastructures and processes from error-prone on-premises to highly available, secure, and efficient cloud, as well as providing cutting edge Internet of Things (IoT), AI/ML, and analytics solutions.

Microsoft Azure holds the promise of the fourth industrial revolution by providing manufacturing solutions that can:

- help build more agile smart factories with industrial IoT,
- create more resilient and profitable supply chains,
- transform your work force productivity,
- unlock innovation and new business models, and
- engage with customers in new ways.

To learn how you can modernize your manufacturing business using Azure, visit Azure for manufacturing. For additional resources, see Microsoft Trusted Cloud for Manufacturing.

Architectures for manufacturing

The following articles provide detailed analysis of architectures developed and recommended for the manufacturing industry.

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Azure Industrial IoT Analytics Guidance	Build an architecture for an Industrial IoT (IIoT) analytics solution on Azure using PaaS (Platform as a service) components.	IoT
Upscale machine learning lifecycle with MLOps framework	Learn how a Fortune 500 food company improved its demand forecasting and optimized the product stocks in different stores across several regions in US with the help of customized machine learning models.	AI/ML
Speech-to-text transcription pipeline with Azure Cognitive Services	Improve the efficiency of your customer care centers and transform your business by analyzing high volumes of recorded calls and building a speech-to-text transcription pipeline with Azure Cognitive Services.	AI/ML

Solution ideas for manufacturing

The following are other ideas that you can use as a starting point for your manufacturing solution.

- Supply Chain Track and Trace
- Knowledge mining for customer support and feedback analysis
- Defect prevention with predictive maintenance
- Image classification with Convolutional Neural Networks
- Create personalized marketing solutions in near real time
- Predictive Maintenance
- Predictive Marketing with Machine Learning
- Quality Assurance
- Anomaly Detector Process
- Demand Forecasting
- Demand forecasting for shipping and distribution
- Predictive Aircraft Engine Monitoring

Azure industrial IoT analytics guidance

11/19/2021 • 11 minutes to read • Edit Online

This article series shows a recommended architecture for an Industrial IoT (IIoT) *analytics solution* on Azure using PaaS (Platform as a service) components. Industrial IoT or IIoT is the application of *Internet of Things* in the manufacturing industry.

An IIoT analytics solution can be used to build a variety of applications that provide:

- Asset monitoring
- Process dashboards
- Overall equipment effectiveness (OEE)
- Predictive maintenance
- Forecasting

Such an IIoT analytics solution relies on real-time and historical data from industrial devices and control systems located in discrete and process manufacturing facilities. These include PLCs (Programmable Logic Controller), industrial equipment, SCADA (Supervisory Control and Data Acquisition) systems, MES (Manufacturing Execution System), and Process Historians. The architecture covered by this series, includes guidance for connecting to all these systems.

A modern IIoT analytics solution goes beyond moving existing industrial processes and tools to the cloud. It involves transforming your operations and processes, embracing PaaS services, and leveraging the power of machine learning and the intelligent edge to optimize industrial processes.

The following list shows some typical *personas* who would use the solution and how they would use this solution:

- **Plant Manager:** responsible for the entire operations, production, and administrative tasks of the manufacturing plant.
- **Production Manager:** responsible for production of a certain number of components.
- **Process Engineer:** responsible for designing, implementing, controlling, and optimizing industrial processes.
- **Operations Manager:** responsible for overall efficiency of operation in terms of cost reduction, process time, process improvement, and so on.
- **Data Scientist:** responsible for building and training predictive Machine Learning models using historical industrial telemetry.

The following architecture diagram shows the core subsystems that form an IIoT analytics solution.

> **NOTE**
>
> This architecture represents an ingestion-only pattern. No control commands are sent back to the industrial systems or devices.

The architecture consists of a number of subsystems and services, and makes use of the Azure Industrial IoT components. Your own solution may not use all these services or may have additional services. This architecture also lists alternative service options, where applicable.

> **IMPORTANT**
>
> This architecture includes some services marked as "Preview" or "Public Preview". Preview services are governed by Supplemental Terms of Use for Microsoft Azure Previews.

Industrial systems and devices

In process manufacturing, industrial equipment (for example, flow monitors, pumps, and so on) is often geographically dispersed and must be monitored remotely. Remote Terminal Units (RTUs) connect remote equipment to a central SCADA system. RTUs work well in conditions where connectivity is intermittent, and no reliable continuous power supply exists.

In discrete manufacturing, industrial equipment (for example, factory robots, conveyor systems, and so on) are connected and controlled by a PLC. One or more PLCs may be connected to a central SCADA system using protocols such as Modbus or other industrial protocols.

In some cases, the data from SCADA systems is forwarded and centralized in an MES or a *historian* software (also known as *process historian* or *operational historian*). These historians are often located in IT controlled networks and have some access to the internet.

Frequently, industrial equipment and SCADA systems are in a closed Process Control Network (PCN), behind one or more firewalls, with no direct access to the internet. Historians often contain industrial data from multiple facilities and are located outside of a PCN. So, connecting to a historian is often the path of least resistance, rather than connecting to a SCADA, MES, or PLC. If a historian is not available, then connecting to an MES or SCADA system would be the next logical choice.

The connection to the historian, MES, or SCADA system will depend on what protocols are available on that system. Many systems now include Industry 4.0 standards such as OPC UA. Older systems may only support legacy protocols such as Modbus, ODBC, or SOAP. If so, you will most likely require a protocol translation software running on an *intelligent edge* device.

Intelligent edge

Intelligent edge devices perform some data processing on the device itself or on a field gateway. In most industrial scenarios, the industrial equipment cannot have additional software installed on it. This is why a field gateway is required to connect the industrial equipment to the cloud.

Azure IoT Edge

To connect industrial equipment and systems to the cloud, we recommend using Azure IoT Edge as the field gateway for:

- Protocol and identity translation;
- Edge processing and analytics; and
- Adhering to network security policies (ISA 95, ISA 99).

Azure IoT Edge is a free, open source field gateway software that runs on a variety of supported hardware devices or a virtual machine.

IoT Edge allows you to run edge workloads as Docker container modules. The modules can be developed in several languages, with SDKs provided for Python, Node.js, C#, Java and C. Prebuilt Azure IoT Edge modules from Microsoft and third-party partners are available from the Azure IoT Edge Marketplace.

Real-time industrial data is encrypted and streamed through Azure IoT Edge to Azure IoT Hub using AMQP 1.0 or MQTT 3.1.1 protocols. IoT Edge can operate in offline or intermittent network conditions providing ***store and forward*** capabilities.

There are two system modules provided as part of IoT Edge runtime.

- The **EdgeAgent** module is responsible for pulling down the container orchestration specification (manifest) from the cloud, so that it knows which modules to run. Module configuration is provided as part of the module twin.
- The **EdgeHub** module manages the communication from the device to Azure IoT Hub, as well as the inter-module communication. Messages are routed from one module to the next using JSON configuration.

Azure IoT Edge automatic deployments can be used to specify a standing configuration for new or existing devices. This provides a single location for deployment configuration across thousands of Azure IoT Edge devices.

A number of third-party IoT Edge gateway devices are available from the *Azure Certified for IoT Device Catalog*.

> **IMPORTANT**
> Proper hardware sizing of an IoT Edge gateway is important to ensure edge module performance. See the performance considerations for this architecture.

Gateway patterns

There are three patterns for connecting your devices to Azure via an IoT Edge field gateway (or virtual machine):

1. **Transparent:** Devices already have the capability to send messages to IoT Hub using AMQP or MQTT. Instead of sending the messages directly to the hub, they instead send the messages to IoT Edge, which in turn passes them on to IoT Hub. Each device has an identity and device twin in Azure IoT Hub.

2. **Protocol Translation:** Also known as an opaque gateway pattern. This pattern is often used to connect older brownfield equipment (for example, Modbus) to Azure. Modules are deployed to Azure IoT Edge to perform the protocol conversion. Devices must provide a unique identifier to the gateway.

3. **Identity Translation:** In this pattern, devices cannot communicate directly to IoT Hub (for example, OPC UA Pub/Sub, BLE devices). The gateway is smart enough to understand the protocol used by the downstream devices, provide them identity, and translate IoT Hub primitives. Each device has an identity and device twin in Azure IoT Hub.

Although you can use any of these patterns in your IIoT Analytics Solution, your choice will be driven by which protocol is installed on your industrial systems. For example, if your SCADA system supports ethernet/IP, you will need to use a protocol translation software to convert ethernet/IP to MQTT or AMQP. See the Connecting to Historians section for additional guidance.

IoT Edge gateways can be provisioned at scale using the Azure IoT Hub Device Provisioning Service (DPS). DPS is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention, enabling customers to provision millions of devices in a secure and scalable manner.

OPC UA

OPC UA is the successor to OPC Classic (OPC DA, AE, HDA). The OPC UA standard is maintained by the OPC Foundation. Microsoft has been a member of the OPC Foundation since 1996 and has supported OPC UA on Azure since 2016.

Industry and domain-specific *Information Models* can be created based on the OPC UA *Data Model*. The specifications of such Information Models (also called *industry standard models* since they typically address a dedicated industry problem) are called Companion Specifications. The synergy of the OPC UA infrastructure to exchange such industry information models enables interoperability at the semantic level. OPC UA can use a number of transport protocols including MQTT, AMQP, and UADP.

Microsoft has developed open source Azure Industrial IoT components, based on OPC UA, which implement identity translation pattern:

- OPC Twin consists of microservices and an Azure IoT Edge module to connect the cloud and the factory network. OPC Twin provides discovery, registration, and synchronous remote control of industrial devices through REST APIs.
- OPC Publisher is an Azure IoT Edge module that connects to existing OPC UA servers and publishes telemetry data from OPC UA servers in OPC UA PubSub format, in both JSON and binary.
- OPC Vault is a microservice that can configure, register, and manage certificate lifecycle for OPC UA server and client applications in the cloud.
- Discovery Services is an Azure IoT Edge module that supports network scanning and OPC UA discovery.

The Microsoft Azure IIoT solution also contains a number of services, REST APIs, deployment scripts, and configuration tools that you can integrate into your IIoT analytics solution. These are open source and available on GitHub.

Edge workloads

The ability to run custom or third-party modules at the edge is important.

- If you want to respond to emergencies as quickly as possible, you can run anomaly detection or Machine Learning module in tight control loops at the edge.
- If you want to reduce bandwidth costs and avoid transferring terabytes of raw data, you can clean and aggregate the data locally then send only the insights to the cloud for analysis.
- If you want to convert legacy industrial protocols, you can develop a custom module or purchase a third-party module for protocol translation.
- If you want to quickly respond to an event on the factory floor, you can use an edge module to detect that event and another module to respond to it.

Microsoft and our partners have made available on the Azure Marketplace a number of edge modules, which can be used in your IIoT analytics solution. Protocol and identity translation are the most common edge workloads used within an IIoT analytics solution. In the future, expect to see other workloads such as closed loop control using edge ML models.

Connecting to historians

A common pattern when developing an IIoT analytics solution is to connect to a process historian and stream real-time data from the historian to Azure IoT Hub. How this is done, will depend on which protocols are installed and accessible (that is, not blocked by firewalls) on the historian.

PROTOCOL AVAILABLE ON HISTORIAN	OPTIONS
OPC UA	- Use Azure IoT Edge, along with OPC Publisher, OPC Twin, and OPC Vault, to send OPC UA data over MQTT to IoT Hub. OPC Twin also has support for OPC UA HDA profile, useful for obtaining historical data. - Use a third-party Azure IoT Edge OPC UA module to send OPC UA data over MQTT to IoT Hub.
OPC DA	- Use third-party software to convert OPC DA to OPC UA and send OPC UA data to IoT Hub over MQTT. Or use OPC Publisher, OPC Twin and OPC Vault to send OPC UA data over MQTT to IoT Hub.
Web Service	- Use a custom Azure IoT Edge HTTP module to poll the web service. - Use third-party software that supports HTTP to MQTT 3.1.1 or AMQP 1.0.
MQTT 3.1.1 (Can publish MQTT messages)	- Connect historian directly to Azure IoT Hub using MQTT. - Connect historian to Azure IoT Edge as a leaf device. See Transparent Gateway pattern.

PROTOCOL AVAILABLE ON HISTORIAN	OPTIONS
Other	- Use a custom Azure IoT Edge module. - Use third-party software to convert to MQTT 3.1.1 or AMQP 1.0.

A number of Microsoft partners have developed protocol and identity translation modules or solutions that are available on the Azure Marketplace.

Some historian vendors also provide first-class capabilities to send data to Azure.

HISTORIAN	OPTIONS
OSIsoft PI	PI Integrator for Azure
Honeywell	Uniformance Cloud Historian

Once real time data streaming has been established between your historian and Azure IoT Hub, it is important to export your historian's historical data and import it into your IIoT analytics solution. For guidance on how to accomplish this, see Historical Data Ingestion.

Cloud gateway

A cloud gateway provides a cloud hub for devices and field gateways to connect securely to the cloud and send data. It also provides device management capabilities. For the cloud gateway, we recommend Azure IoT Hub. IoT Hub is a hosted cloud service that ingests events from devices and IoT Edge gateways. IoT Hub provides secure connectivity, event ingestion, bidirectional communication, and device management. When IoT Hub is combined with the Azure Industrial IoT components, you can control your industrial devices using cloud-based REST APIs.

IoT Hub supports the following protocols:

- MQTT 3.1.1,
- MQTT over WebSockets,
- AMQP 1.0,
- AMQP over WebSockets, and
- HTTPS.

If the industrial device or system supports any of these protocols, it can send data directly to IoT Hub. In most industrial environments, this is not permissible because of PCN firewalls and network security policies (ISA 95, ISA 99). In such cases, an Azure IoT Edge field gateway can be installed in a DMZ between the PCN and the Internet.

Next steps

To learn the services recommended for this architecture, continue reading the series with Services in an IIoT analytics solution.

Solutions for the media and entertainment industry

11/19/2021 • 2 minutes to read • Edit Online

The media and entertainment industry captures one of the largest market shares. It is comprised of businesses that produce and distribute content, such as motion pictures, television programs and commercials, streaming content, music and audio recordings, radio, book publishing, video games, and so on. With the COVID-19 pandemic greatly impacting and accelerating shifts in consumer behaviors, this industry is seeing trends such as creating more virtual, streamed, and personal content. It is all the more important for media businesses to harness the power of cloud computing and reach their customers in more personalized and innovative ways.

Microsoft's Azure and other offerings are committed to empower media and entertainment businesses to achieve more:

- accelerate content creation,
- provide cost-effective content management platforms,
- optimize and personalize content delivery,
- modernize collaboration.

To learn how Azure can provide an intelligent cloud backbone to content owners and creators, visit Azure for media and entertainment. Microsoft offerings are transforming and empowering media businesses; see some case studies at Intelligent Media and Entertainment.

Architectures for media and entertainment

The following articles provide detailed analysis of architectures developed and recommended for the media and entertainment industry.

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
3D video rendering with Azure Batch	Use Azure Batch as a powerful yet cost-effective alternative to expensive high end computing resources, for 3D video rendering.	Compute
Digital image-based modeling on Azure	Build a high-performance and scalable image-based modeling architecture using Azure infrastructure-as-a-service (IaaS).	Compute
Image classification on Azure	Quickly build an architecture to process, classify, and analyze images, using Azure Computer Vision API.	AI/ML
Movie recommendations on Azure	Automate movie and product recommendations by using an Azure Data Science Virtual Machine to train an Azure Machine Learning model.	AI/ML

ARCHITECTURE	SUMMARY	TECHNOLOGY FOCUS
Scalable personalization on Azure	Build a *content-based* personalized recommendation system for customers by learning their preferences from past purchases and interactions.	AI/ML
Analyze news feeds with near real-time analytics	Build a pipeline for mass ingestion and near real-time analysis of documents coming from public RSS news feeds using Azure services.	Analytics

Solution ideas for media and entertainment

The following are other ideas that you can use as a starting point for your media and entertainment solution.

- Knowledge mining in digital asset management
- HPC Media Rendering
- Suggest content tags with deep learning and NLP
- Keyword search/speech-to-text/OCR digital media

Artificial intelligence architecture

11/19/2021 • 15 minutes to read • Edit Online

Artificial intelligence (AI) is the capability of a computer to imitate intelligent human behavior. Through AI, machines can analyze images, comprehend speech, interact in natural ways, and make predictions using data.

AI concepts

Algorithm

An *algorithm* is a sequence of calculations and rules used to solve a problem or analyze a set of data. It is like a flow chart, with step-by-step instructions for questions to ask, but written in math and programming code. An algorithm may describe how to determine whether a pet is a cat, dog, fish, bird, or lizard. Another far more complicated algorithm may describe how to identify a written or spoken language, analyze its words, translate them into a different language, and then check the translation for accuracy.

Machine learning

Machine learning (ML) is an AI technique that uses mathematical algorithms to create predictive models. An algorithm is used to parse data fields and to "learn" from that data by using patterns found within it to generate models. Those models are then used to make informed predictions or decisions about new data.

The predictive models are validated against known data, measured by performance metrics selected for specific business scenarios, and then adjusted as needed. This process of learning and validation is called *training*. Through periodic retraining, ML models are improved over time.

- Machine learning at scale
- What are the machine learning products at Microsoft?

Deep learning

Deep learning is a type of ML that can determine for itself whether its predictions are accurate. It also uses algorithms to analyze data, but it does so on a larger scale than ML.

Deep learning uses artificial neural networks, which consist of multiple layers of algorithms. Each layer looks at

the incoming data, performs its own specialized analysis, and produces an output that other layers can understand. This output is then passed to the next layer, where a different algorithm does its own analysis, and so on.

With many layers in each neural network-and sometimes using multiple neural networks-a machine can learn through its own data processing. This requires much more data and much more computing power than ML.

- Deep learning versus machine learning
- Distributed training of deep learning models on Azure
- Batch scoring of deep learning models on Azure
- Training of Python scikit-learn and deep learning models on Azure
- Real-time scoring of Python scikit-learn and deep learning models on Azure

Bots

A *bot* is an automated software program designed to perform a particular task. Think of it as a robot without a body. Early bots were comparatively simple, handling repetitive and voluminous tasks with relatively straightforward algorithmic logic. An example would be web crawlers used by search engines to automatically explore and catalog web content.

Bots have become much more sophisticated, using AI and other technologies to mimic human activity and decision-making, often while interacting directly with humans through text or even speech. Examples include bots that can take a dinner reservation, chatbots (or conversational AI) that help with customer service interactions, and social bots that post breaking news or scientific data to social media sites.

Microsoft offers the Azure Bot Service, a managed service purpose-built for enterprise-grade bot development.

- About Azure Bot Service
- Ten guidelines for responsible bots
- Azure reference architecture: Enterprise-grade conversational bot
- Example workload: Conversational chatbot for hotel reservations on Azure

Autonomous systems

Autonomous systems are part of an evolving new class that goes beyond basic automation. Instead of performing a specific task repeatedly with little or no variation (like bots do), autonomous systems bring intelligence to machines so they can adapt to changing environments to accomplish a desired goal.

Smart buildings use autonomous systems to automatically control operations like lighting, ventilation, air conditioning, and security. A more sophisticated example would be a self-directed robot exploring a collapsed mine shaft to thoroughly map its interior, determine which portions are structurally sound, analyze the air for breathability, and detect signs of trapped miners in need of rescue-all without a human monitoring in real time on the remote end.

- Autonomous systems and solutions from Microsoft AI

General info on Microsoft AI

Learn more about Microsoft AI, and keep up-to-date with related news:

- Microsoft AI School
- Azure AI platform page
- Microsoft AI platform page
- Microsoft AI Blog

- Microsoft AI on GitHub: Samples, reference architectures, and best practices
- Azure Architecture Center

High-level architectural types

Prebuilt AI

Prebuilt AI is exactly what it sounds like-off-the-shelf AI models, services, and APIs that are ready to use. These help you add intelligence to apps, websites, and flows without having to gather data and then build, train, and publish your own models.

One example of prebuilt AI might be a pretrained model that can be incorporated as is or used to provide a baseline for further custom training. Another example would be a cloud-based API service that can be called at will to process natural language in a desired fashion.

Azure Cognitive Services

Cognitive Services provide developers the opportunity to use prebuilt APIs and integration toolkits to create applications that can see, hear, speak, understand, and even begin to reason. The catalog of services within Cognitive Services can be categorized into five main pillars: Vision, Speech, Language, Web Search, and Decision/Recommendation.

- Azure Cognitive Services documentation
- Try Azure Cognitive Services for free
- Choosing a Microsoft Cognitive Services technology
- Choosing a natural language processing technology in Azure

Prebuilt AI models in AI Builder

AI Builder is a new capability in Microsoft Power Platform that provides a point-and-click interface for adding AI to your apps, even if you have no coding or data science skills. (Some features in AI Builder have not yet released for general availability and remain in preview status. For more information, refer to the Feature availability by region page.)

You can build and train your own models, but AI Builder also provides select prebuilt AI models that are ready for use right away. For example, you can add a component in Microsoft Power Apps based on a prebuilt model that recognizes contact information from business cards.

- Power Apps on Azure
- AI Builder documentation
- AI model types in AI Builder
- Overview of prebuilt AI models in AI Builder

Custom AI

Although prebuilt AI is useful (and increasingly flexible), the best way to get what you need from AI is probably to build a system yourself. This is obviously a very deep and complex subject, but let's look at some basic concepts beyond what we've just covered.

Code languages

The core concept of AI is the use of algorithms to analyze data and generate models to describe (or *score*) it in ways that are useful. Algorithms are written by developers and data scientists (and sometimes by other algorithms) using programming code. Two of the most popular programming languages for AI development are currently Python and R.

Python is a general-purpose, high-level programming language. It has a simple, easy-to-learn syntax that

emphasizes readability. There is no compiling step. Python has a large standard library, but it also supports the ability to add modules and packages. This encourages modularity and lets you expand capabilities when needed. There is a large and growing ecosystem of AI and ML libraries for Python, including many that are readily available in Azure.

- Python on Azure product home page
- Azure for Python developers
- Azure Machine Learning SDK for Python
- Introduction to machine learning with Python and Azure Notebooks
- scikit-learn. An open-source ML library for Python
- PyTorch. An open-source Python library with a rich ecosystem that can be used for deep learning, computer vision, natural language processing, and more
- TensorFlow. An open-source symbolic math library also used for ML applications and neural networks
- Tutorial: Apply machine learning models in Azure Functions with Python and TensorFlow

R is a language and environment for statistical computing and graphics. It can be used for everything from mapping broad social and marketing trends online to developing financial and climate models.

Microsoft has fully embraced the R programming language and provides many different options for R developers to run their code in Azure.

- R developer's guide to Azure
- Microsoft R Open. An enhanced distribution of R from Microsoft, fully compatible with R-3.5.3, with additional capabilities for improved performance and reproducibility, in addition to support for Windows- and Linux-based platforms
- Tutorial: Create a logistic regression model in R with Azure Machine Learning

Training

Training is core to machine learning. It is the iterative process of "teaching" an algorithm to create models, which are used to analyze data and then make accurate predictions from it. In practice, this process has three general phases: training, validation, and testing.

During the training phase, a quality set of known data is tagged so that individual fields are identifiable. The tagged data is fed to an algorithm configured to make a particular prediction. When finished, the algorithm outputs a model that describes the patterns it found as a set of parameters. During validation, fresh data is tagged and used to test the model. The algorithm is adjusted as needed and possibly put through more training. Finally, the testing phase uses real-world data without any tags or preselected targets. Assuming the model's results are accurate, it is considered ready for use and can be deployed.

- Train models with Azure Machine Learning

Hyperparameter tuning

Hyperparameters are data variables that govern the training process itself. They are configuration variables that control how the algorithm operates. Hyperparameters are thus typically set before model training begins and are not modified within the training process in the way that parameters are. Hyperparameter tuning involves running trials within the training task, assessing how well they are getting the job done, and then adjusting as needed. This process generates multiple models, each trained using different families of hyperparameters.

- Tune hyperparameters for your model with Azure Machine Learning

Model selection

The process of training and hyperparameter tuning produces numerous candidate models. These can have

many different variances, including the effort needed to prepare the data, the flexibility of the model, the amount of processing time, and of course the degree of accuracy of its results. Choosing the best trained model for your needs and constraints is called *model selectio*n, but this is as much about preplanning before training as it is about choosing the one that works best.

Automated machine learning (AutoML)

Automated machine learning, also known as AutoML, is the process of automating the time-consuming, iterative tasks of machine learning model development. It can significantly reduce the time it takes to get production-ready ML models. Automated ML can assist with model selection, hyperparameter tuning, model training, and other tasks, without requiring extensive programming or domain knowledge.

- What is automated machine learning?

Scoring

Scoring is also called *prediction* and is the process of generating values based on a trained machine learning model, given some new input data. The values, or scores, that are created can represent predictions of future values, but they might also represent a likely category or outcome. The scoring process can generate many different types of values:

- A list of recommended items and a similarity score
- Numeric values, for time series models and regression models
- A probability value, indicating the likelihood that a new input belongs to some existing category
- The name of a category or cluster to which a new item is most similar
- A predicted class or outcome, for classification models

Batch scoring is when data is collected during some fixed period of time and then processed in a batch. This might include generating business reports or analyzing customer loyalty.

Real-time scoring is exactly that-scoring that is ongoing and performed as quickly as possible. The classic example is credit card fraud detection, but real-time scoring can also be used in speech recognition, medical diagnoses, market analyses, and many other applications.

General info on custom AI on Azure

- Microsoft AI on GitHub: Samples, reference architectures, and best practices
- Custom AI on Azure GitHub repo. A collection of scripts and tutorials to help developers effectively use Azure for their AI workloads
- Azure Machine Learning SDK for Python
- Azure Machine Learning service example notebooks (Python). A GitHub repo of example notebooks demonstrating the Azure Machine Learning Python SDK
- Azure Machine Learning SDK for R

Azure AI platform offerings

Following is a breakdown of Azure technologies, platforms, and services you can use to develop AI solutions for your needs.

Azure Machine Learning

This is an enterprise-grade machine learning service to build and deploy models faster. Azure Machine Learning offers web interfaces and SDKs so you can quickly train and deploy your machine learning models and pipelines at scale. Use these capabilities with open-source Python frameworks, such as PyTorch, TensorFlow, and scikit-learn.

- What are the machine learning products at Microsoft?
- Azure Machine Learning product home page
- Azure Machine Learning Data Architecture Guide overview
- Azure Machine Learning documentation overview
- What is Azure Machine Learning? General orientation with links to many learning resources, SDKs, documentation, and more

Machine learning reference architectures for Azure

- Training of Python scikit-learn and deep learning models on Azure
- Distributed training of deep learning models on Azure
- Batch scoring of Python machine learning models on Azure
- Batch scoring of deep learning models on Azure
- Real-time scoring of Python scikit-learn and deep learning models on Azure
- Machine learning operationalization (MLOps) for Python models using Azure Machine Learning
- Batch scoring of R machine learning models on Azure
- Real-time scoring of R machine learning models on Azure
- Batch scoring of Spark machine learning models on Azure Databricks
- Enterprise-grade conversational bot
- Build a real-time recommendation API on Azure

Azure automated machine learning

Azure provides extensive support for automated ML. Developers can build models using a no-code UI or through a code-first notebooks experience.

- Azure automated machine learning product home page
- Azure automated ML infographic (PDF)
- Tutorial: Create a classification model with automated ML in Azure Machine Learning
- Tutorial: Use automated machine learning to predict taxi fares
- Configure automated ML experiments in Python
- Use the CLI extension for Azure Machine Learning
- Automate machine learning activities with the Azure Machine Learning CLI

Azure Cognitive Services

This is a comprehensive family of AI services and cognitive APIs to help you build intelligent apps. These domain-specific, pretrained AI models can be customized with your data.

- Cognitive Services product home page
- Azure Cognitive Services documentation

Azure Cognitive Search

This is an AI-powered cloud search service for mobile and web app development. The service can search over private heterogenous content, with options for AI enrichment if your content is unstructured or unsearchable in

raw form.

- Azure Cognitive Search product home page
- Getting started with AI enrichment
- Azure Cognitive Search documentation overview
- Choosing a natural language processing technology in Azure
- Quickstart: Create an Azure Cognitive Search cognitive skill set in the Azure portal

Azure Bot Service

This is a purpose-built bot development environment with out-of-the-box templates to get started quickly.

- Azure Bot Service product home page
- Azure Bot Service documentation overview
- Azure reference architecture: Enterprise-grade conversational bot
- Example workload: Conversational chatbot for hotel reservations on Azure
- Microsoft Bot Framework
- GitHub Bot Builder repo

Apache Spark on Azure

Apache Spark is a parallel processing framework that supports in-memory processing to boost the performance of big data analytic applications. Spark provides primitives for in-memory cluster computing. A Spark job can load and cache data into memory and query it repeatedly, which is much faster than disk-based applications, such as Hadoop.

Apache Spark in Azure HDInsight is the Microsoft implementation of Apache Spark in the cloud. Spark clusters in HDInsight are compatible with Azure Storage and Azure Data Lake Storage, so you can use HDInsight Spark clusters to process your data stored in Azure.

The Microsoft Machine Learning library for Apache Spark is MMLSpark (Microsoft ML for Apache Spark). It is an open-source library that adds many deep learning and data science tools, networking capabilities, and production-grade performance to the Spark ecosystem. Learn more about MMLSpark features and capabilities.

- Azure HDInsight overview. Basic information about features, cluster architecture, and use cases, with pointers to quickstarts and tutorials.
- Tutorial: Build an Apache Spark machine learning application in Azure HDInsight
- Apache Spark best practices on HDInsight
- Configure HDInsight Apache Spark Cluster settings
- Machine learning on HDInsight
- GitHub repo for MMLSpark: Microsoft Machine Learning library for Apache Spark
- Create an Apache Spark machine learning pipeline on HDInsight

Azure Databricks Runtime for Machine Learning

Azure Databricks is an Apache Spark–based analytics platform with one-click setup, streamlined workflows, and an interactive workspace for collaboration between data scientists, engineers, and business analysts.

Databricks Runtime for Machine Learning (Databricks Runtime ML) lets you start a Databricks cluster with all of the libraries required for distributed training. It provides a ready-to-go environment for machine learning and

data science. Plus, it contains multiple popular libraries, including TensorFlow, PyTorch, Keras, and XGBoost. It also supports distributed training using Horovod.

- Azure Databricks product home page
- Azure Databricks documentation
- Machine learning capabilities in Azure Databricks
- How-to guide: Databricks Runtime for Machine Learning
- Batch scoring of Spark machine learning models on Azure Databricks
- Deep learning overview for Azure Databricks

Customer stories

Different industries are applying AI in innovative and inspiring ways. Following are a number of customer case studies and success stories:

- ASOS: Online retailer solves challenges with Azure Machine Learning service
- KPMG helps financial institutions save millions in compliance costs with Azure Cognitive Services
- Volkswagen: Machine translation speaks Volkswagen – in 40 languages
- Buncee: NYC school empowers readers of all ages and abilities with Azure AI
- InterSystems: Data platform company boosts healthcare IT by generating critical information at unprecedented speed
- Zencity: Data-driven startup uses funding to help local governments support better quality of life for residents
- Bosch uses IoT innovation to drive traffic safety improvements by helping drivers avoid serious accidents
- Automation Anywhere: Robotic process automation platform developer enriches its software with Azure Cognitive Services
- Wix deploys smart, scalable search across 150 million websites with Azure Cognitive Search
- Asklepios Klinik Altona: Precision surgeries with Microsoft HoloLens 2 and 3D visualization
- AXA Global P&C: Global insurance firm models complex natural disasters with cloud-based HPC

Browse more AI customer stories

Next steps

- To learn about the artificial intelligence development products available from Microsoft, refer to the Microsoft AI platform page.
- For training in how to develop AI solutions, refer to Microsoft AI School.
- Microsoft AI on GitHub: Samples, reference architectures, and best practices organizes the Microsoft open source AI-based repositories, providing tutorials and learning materials.

Choose a Microsoft cognitive services technology

11/19/2021 • 3 minutes to read • Edit Online

Microsoft cognitive services are cloud-based APIs that you can use in artificial intelligence (AI) applications and data flows. They provide you with pretrained models that are ready to use in your application, requiring no data and no model training on your part. The cognitive services are developed by Microsoft's AI and Research team and leverage the latest deep learning algorithms. They are consumed over HTTP REST interfaces. In addition, SDKs are available for many common application development frameworks.

The cognitive services include:

- Text analysis
- Computer vision
- Video analytics
- Speech recognition and generation
- Natural language understanding
- Intelligent search

Key benefits:

- Minimal development effort for state-of-the-art AI services.
- Easy integration into apps via HTTP REST interfaces.
- Built-in support for consuming cognitive services in Azure Data Lake Analytics.

Considerations:

- Only available over the web. Internet connectivity is generally required. An exception is the Custom Vision Service, whose trained model you can export for prediction on devices and at the IoT edge.
- Although considerable customization is supported, the available services may not suit all predictive analytics requirements.

What are your options when choosing amongst the cognitive services?

In Azure, there are dozens of Cognitive Services available. The current listing of these is available in a directory categorized by the functional area they support:

- Vision
- Speech
- Decision
- Search
- Language

Key selection criteria

To narrow the choices, start by answering these questions:

- What type of data are you dealing with? Narrow your options based on the type of input data you are working with. For example, if your input is text, select from the services that have an input type of text.

- Do you have the data to train a model? If yes, consider the custom services that enable you to train their underlying models with data that you provide, for improved accuracy and performance.

Capability matrix

The following tables summarize the key differences in capabilities.

Uses prebuilt models

CAPABILITY	INPUT TYPE	KEY BENEFIT
Text Analytics API	Text	Evaluate sentiment and topics to understand what users want.
Entity Linking API	Text	Power your app's data links with named entity recognition and disambiguation.
Language Understanding Intelligent Service (LUIS)	Text	Teach your apps to understand commands from your users.
QnA Maker Service	Text	Distill FAQ formatted information into conversational, easy-to-navigate answers.
Linguistic Analysis API	Text	Simplify complex language concepts and parse text.
Knowledge Exploration Service	Text	Enable interactive search experiences over structured data via natural language inputs.
Web Language Model API	Text	Use predictive language models trained on web-scale data.
Academic Knowledge API	Text	Tap into the wealth of academic content in the Microsoft Academic Graph populated by Bing.
Bing Autosuggest API	Text	Give your app intelligent autosuggest options for searches.
Bing Spell Check API	Text	Detect and correct spelling mistakes in your app.
Translator Text API	Text	Machine translation.
Recommendations API	Text	Predict and recommend items your customers want.
Bing Entity Search API	Text (web search query)	Identify and augment entity information from the web.
Bing Image Search API	Text (web search query)	Search for images.
Bing News Search API	Text (web search query)	Search for news.

CAPABILITY	INPUT TYPE	KEY BENEFIT
Bing Video Search API	Text (web search query)	Search for videos.
Bing Web Search API	Text (web search query)	Get enhanced search details from billions of web documents.
Bing Speech API	Text or Speech	Convert speech to text and back again.
Speaker Recognition API	Speech	Use speech to identify and authenticate individual speakers.
Translator Speech API	Speech	Perform real-time speech translation.
Computer Vision API	Images (or frames from video)	Distill actionable information from images, automatically create description of photos, derive tags, recognize celebrities, extract text, and create accurate thumbnails.
Content Moderator	Text, Images or Video	Automated image, text, and video moderation.
Emotion API	Images (photos with human subjects)	Identify the range emotions of human subjects.
Face API	Images (photos with human subjects)	Detect, identify, analyze, organize, and tag faces in photos.
Video Indexer	Video	Video insights such as sentiment, transcript speech, translate speech, recognize faces and emotions, and extract keywords.

Trained with custom data you provide

CAPABILITY	INPUT TYPE	KEY BENEFIT
Custom Vision Service	Images (or frames from video)	Customize your own computer vision models.
Custom Speech Service	Speech	Overcome speech recognition barriers like speaking style, background noise, and vocabulary.
Custom Decision Service	Web content (for example, RSS feed)	Use machine learning to automatically select the appropriate content for your home page
Bing Custom Search API	Text (web search query)	Commercial-grade search tool.

Compare the machine learning products and technologies from Microsoft

11/19/2021 • 8 minutes to read • Edit Online

Learn about the machine learning products and technologies from Microsoft. Compare options to help you choose how to most effectively build, deploy, and manage your machine learning solutions.

Cloud-based machine learning products

The following options are available for machine learning in the Azure cloud.

CLOUD OPTIONS	WHAT IT IS	WHAT YOU CAN DO WITH IT
Azure Machine Learning	Managed platform for machine learning	Use a pretrained model. Or, train, deploy, and manage models on Azure using Python and CLI
Azure Cognitive Services	Pre-built AI capabilities implemented through REST APIs and SDKs	Build intelligent applications quickly using standard programming languages. Doesn't require machine learning and data science expertise
Azure SQL Managed Instance Machine Learning Services	In-database machine learning for SQL	Train and deploy models inside Azure SQL Managed Instance
Machine learning in Azure Synapse Analytics	Analytics service with machine learning	Train and deploy models inside Azure SQL Managed Instance
Machine learning and AI with ONNX in Azure SQL Edge	Machine learning in SQL on IoT	Train and deploy models inside Azure SQL Edge
Azure Databricks	Apache Spark-based analytics platform	Build and deploy models and data workflows using integrations with open-source machine learning libraries and the MLFlow platform.

On-premises machine learning products

The following options are available for machine learning on-premises. On-premises servers can also run in a virtual machine in the cloud.

ON-PREMISES OPTIONS	WHAT IT IS	WHAT YOU CAN DO WITH IT
SQL Server Machine Learning Services	In-database machine learning for SQL	Train and deploy models inside SQL Server
Machine Learning Services on SQL Server Big Data Clusters	Machine learning in Big Data Clusters	Train and deploy models on SQL Server Big Data Clusters

Development platforms and tools

The following development platforms and tools are available for machine learning.

PLATFORMS/TOOLS	WHAT IT IS	WHAT YOU CAN DO WITH IT
Azure Data Science Virtual Machine	Virtual machine with pre-installed data science tools	Develop machine learning solutions in a pre-configured environment
ML.NET	Open-source, cross-platform machine learning SDK	Develop machine learning solutions for .NET applications
Windows ML	Windows 10 machine learning platform	Evaluate trained models on a Windows 10 device
MMLSpark	Open-source, distributed, machine learning and microservices framework for Apache Spark	Create and deploy scalable machine learning applications for Scala and Python.
Machine Learning extension for Azure Data Studio	Open-source and cross-platform machine learning extension for Azure Data Studio	Manage packages, import machine learning models, make predictions, and create notebooks to run experiments for your SQL databases

Azure Machine Learning

Azure Machine Learning is a fully managed cloud service used to train, deploy, and manage machine learning models at scale. It fully supports open-source technologies, so you can use tens of thousands of open-source Python packages such as TensorFlow, PyTorch, and scikit-learn. Rich tools are also available, such as Compute instances, Jupyter notebooks, or the Azure Machine Learning for Visual Studio Code extension, a free extension that allows you to manage your resources, model training workflows and deployments in Visual Studio Code. Azure Machine Learning includes features that automate model generation and tuning with ease, efficiency, and accuracy.

Use Python SDK, Jupyter notebooks, R, and the CLI for machine learning at cloud scale. For a low-code or no-code option, use Azure Machine Learning's interactive designer in the studio to easily and quickly build, test, and deploy models using pre-built machine learning algorithms.

Try Azure Machine Learning for free.

Type	Cloud-based machine learning solution
Supported languages	Python, R
Machine learning phases	Model training Deployment MLOps/Management
Key benefits	Code first (SDK) and studio & drag-and-drop designer web interface authoring options. Central management of scripts and run history, making it easy to compare model versions. Easy deployment and management of models to the cloud or edge devices.

Considerations	Requires some familiarity with the model management model.

Azure Cognitive Services

Azure Cognitive Services is a set of *pre-built* APIs that enable you to build apps that use natural methods of communication. The term pre-built suggests that you do not need to bring datasets or data science expertise to train models to use in your applications. That's all done for you and packaged as APIs and SDKs that allow your apps to see, hear, speak, understand, and interpret user needs with just a few lines of code. You can easily add intelligent features to your apps, such as:

- **Vision:** Object detection, face recognition, OCR, etc. See Computer Vision, Face, Form Recognizer.
- **Speech:** Speech-to-text, text-to-speech, speaker recognition, etc. See Speech Service.
- **Language:** Translation, Sentiment analysis, key phrase extraction, language understanding, etc. See Translator, Text Analytics, Language Understanding, QnA Maker
- **Decision:** Anomaly detection, content moderation, reinforcement learning. See Anomaly Detector, Content Moderator, Personalizer.

Use Cognitive Services to develop apps across devices and platforms. The APIs keep improving, and are easy to set up.

Type	APIs for building intelligent applications
Supported languages	Various options depending on the service. Standard ones are C#, Java, JavaScript, and Python.
Machine learning phases	Deployment
Key benefits	Build intelligent applications using pre-trained models available through REST API and SDK. Variety of models for natural communication methods with vision, speech, language, and decision. No machine learning or data science expertise required.

SQL machine learning

SQL machine learning adds statistical analysis, data visualization, and predictive analytics in Python and R for relational data, both on-premises and in the cloud. Current platforms and tools include:

- SQL Server Machine Learning Services
- Machine Learning Services on SQL Server Big Data Clusters
- Azure SQL Managed Instance Machine Learning Services
- Machine learning in Azure Synapse Analytics
- Machine learning and AI with ONNX in Azure SQL Edge
- Machine Learning extension for Azure Data Studio

Use SQL machine learning when you need built-in AI and predictive analytics on relational data in SQL.

Type	On-premises predictive analytics for relational data

Supported languages	Python, R, SQL
Machine learning phases	Data preparation Model training Deployment
Key benefits	Encapsulate predictive logic in a database function, making it easy to include in data-tier logic.
Considerations	Assumes a SQL database as the data tier for your application.

Azure Data Science Virtual Machine

The Azure Data Science Virtual Machine is a customized virtual machine environment on the Microsoft Azure cloud. It is available in versions for both Windows and Linux Ubuntu. The environment is built specifically for doing data science and developing ML solutions. It has many popular data science, ML frameworks, and other tools pre-installed and pre-configured to jump-start building intelligent applications for advanced analytics.

Use the Data Science VM when you need to run or host your jobs on a single node. Or if you need to remotely scale up your processing on a single machine.

Type	Customized virtual machine environment for data science
Key benefits	Reduced time to install, manage, and troubleshoot data science tools and frameworks. The latest versions of all commonly used tools and frameworks are included. Virtual machine options include highly scalable images with GPU capabilities for intensive data modeling.
Considerations	The virtual machine cannot be accessed when offline. Running a virtual machine incurs Azure charges, so you must be careful to have it running only when required.

Azure Databricks

Azure Databricks is an Apache Spark-based analytics platform optimized for the Microsoft Azure cloud services platform. Databricks is integrated with Azure to provide one-click setup, streamlined workflows, and an interactive workspace that enables collaboration between data scientists, data engineers, and business analysts. Use Python, R, Scala, and SQL code in web-based notebooks to query, visualize, and model data.

Use Databricks when you want to collaborate on building machine learning solutions on Apache Spark.

Type	Apache Spark-based analytics platform
Supported languages	Python, R, Scala, SQL

Machine learning phases	Data preparation Data preprocessing Model training Model tuning Model inference Management Deployment

ML.NET

ML.NET is an open-source, and cross-platform machine learning framework. With ML.NET, you can build custom machine learning solutions and integrate them into your .NET applications. ML.NET offers varying levels of interoperability with popular frameworks like TensorFlow and ONNX for training and scoring machine learning and deep learning models. For resource-intensive tasks like training image classification models, you can take advantage of Azure to train your models in the cloud.

Use ML.NET when you want to integrate machine learning solutions into your .NET applications. Choose between the API for a code-first experience and Model Builder or the CLI for a low-code experience.

Type	Open-source cross-platform framework for developing custom machine learning applications with .NET
Languages supported	C#, F#
Machine learning phases	Data preparation Training Deployment
Key benefits	Data science & ML experience not required Use familiar tools (Visual Studio, VS Code) and languages Deploy where .NET runs Extensible Scalable Local-first experience

Windows ML

Windows ML inference engine allows you to use trained machine learning models in your applications, evaluating trained models locally on Windows 10 devices.

Use Windows ML when you want to use trained machine learning models within your Windows applications.

Type	Inference engine for trained models in Windows devices
Languages supported	C#/C++, JavaScript

MMLSpark

Microsoft ML for Apache Spark (MMLSpark) is an open-source library that expands the distributed computing framework Apache Spark. MMLSpark adds many deep learning and data science tools to the Spark ecosystem, including seamless integration of Spark Machine Learning pipelines with Microsoft Cognitive Toolkit (CNTK),

LightGBM, LIME (Model Interpretability), and OpenCV. You can use these tools to create powerful predictive models on any Spark cluster, such as Azure Databricks or Cosmic Spark.

MMLSpark also brings new networking capabilities to the Spark ecosystem. With the HTTP on Spark project, users can embed any web service into their SparkML models. Additionally, MMLSpark provides easy-to-use tools for orchestrating Azure Cognitive Services at scale. For production-grade deployment, the Spark Serving project enables high throughput, submillisecond latency web services, backed by your Spark cluster.

Type	Open-source, distributed machine learning and microservices framework for Apache Spark
Languages supported	Scala 2.11, Java, Python 3.5+, R (beta)
Machine learning phases	Data preparation Model training Deployment
Key benefits	Scalability Streaming + Serving compatible Fault-tolerance
Considerations	Requires Apache Spark

Next steps

- To learn about all the Artificial Intelligence (AI) development products available from Microsoft, see Microsoft AI platform.
- For training in developing AI and Machine Learning solutions with Microsoft, see Microsoft Learn.

Machine learning at scale

11/19/2021 • 3 minutes to read • Edit Online

Machine learning (ML) is a technique used to train predictive models based on mathematical algorithms. Machine learning analyzes the relationships between data fields to predict unknown values.

Creating and deploying a machine learning model is an iterative process:

- Data scientists explore the source data to determine relationships between *features* and predicted *labels*.
- The data scientists train and validate models based on appropriate algorithms to find the optimal model for prediction.
- The optimal model is deployed into production, as a web service or some other encapsulated function.
- As new data is collected, the model is periodically retrained to improve its effectiveness.

Machine learning at scale addresses two different scalability concerns. The first is training a model against large data sets that require the scale-out capabilities of a cluster to train. The second centers on operationalizing the learned model so it can scale to meet the demands of the applications that consume it. Typically this is accomplished by deploying the predictive capabilities as a web service that can then be scaled out.

Machine learning at scale has the benefit that it can produce powerful, predictive capabilities because better models typically result from more data. Once a model is trained, it can be deployed as a stateless, highly performant scale-out web service.

Model preparation and training

During the model preparation and training phase, data scientists explore the data interactively using languages like Python and R to:

- Extract samples from high volume data stores.
- Find and treat outliers, duplicates, and missing values to clean the data.
- Determine correlations and relationships in the data through statistical analysis and visualization.
- Generate new calculated features that improve the predictiveness of statistical relationships.
- Train ML models based on predictive algorithms.
- Validate trained models using data that was withheld during training.

To support this interactive analysis and modeling phase, the data platform must enable data scientists to explore data using a variety of tools. Additionally, the training of a complex machine learning model can require a lot of intensive processing of high volumes of data, so sufficient resources for scaling out the model training is essential.

Model deployment and consumption

When a model is ready to be deployed, it can be encapsulated as a web service and deployed in the cloud, to an edge device, or within an enterprise ML execution environment. This deployment process is referred to as operationalization.

Challenges

Machine learning at scale produces a few challenges:

- You typically need a lot of data to train a model, especially for deep learning models.

- You need to prepare these big data sets before you can even begin training your model.
- The model training phase must access the big data stores. It's common to perform the model training using the same big data cluster, such as Spark, that is used for data preparation.
- For scenarios such as deep learning, not only will you need a cluster that can provide you scale-out on CPUs, but your cluster will need to consist of GPU-enabled nodes.

Machine learning at scale in Azure

Before deciding which ML services to use in training and operationalization, consider whether you need to train a model at all, or if a prebuilt model can meet your requirements. In many cases, using a prebuilt model is just a matter of calling a web service or using an ML library to load an existing model. Some options include:

- Use the web services provided by Azure Cognitive Services.
- Use the pretrained neural network models provided by the Cognitive Toolkit.
- Embed the serialized models provided by Core ML for an iOS app.

If a prebuilt model does not fit your data or your scenario, options in Azure include Azure Machine Learning, HDInsight with Spark MLlib and MMLSpark, Azure Databricks, Cognitive Toolkit, and SQL Machine Learning Services. If you decide to use a custom model, you must design a pipeline that includes model training and operationalization.

For a list of technology choices for ML in Azure, see:

- Choosing a cognitive services technology
- Choosing a machine learning technology
- Choosing a natural language processing technology

Next steps

The following reference architectures show machine learning scenarios in Azure:

- Batch scoring on Azure for deep learning models
- Real-time scoring of Python Scikit-Learn and Deep Learning Models on Azure

Choose a natural language processing technology in Azure

11/19/2021 • 3 minutes to read • Edit Online

Natural language processing (NLP) is used for tasks such as sentiment analysis, topic detection, language detection, key phrase extraction, and document categorization.

NLP can be use to classify documents, such as labeling documents as sensitive or spam. The output of NLP can be used for subsequent processing or search. Another use for NLP is to summarize text by identifying the entities present in the document. These entities can also be used to tag documents with keywords, which enables search and retrieval based on content. Entities might be combined into topics, with summaries that describe the important topics present in each document. The detected topics may be used to categorize the documents for navigation, or to enumerate related documents given a selected topic. Another use for NLP is to score text for sentiment, to assess the positive or negative tone of a document. These approaches use many techniques from natural language processing, such as:

- **Tokenizer**. Splitting the text into words or phrases.
- **Stemming and lemmatization**. Normalizing words so that different forms map to the canonical word with the same meaning. For example, "running" and "ran" map to "run."
- **Entity extraction**. Identifying subjects in the text.
- **Part of speech detection**. Identifying text as a verb, noun, participle, verb phrase, and so on.
- **Sentence boundary detection**. Detecting complete sentences within paragraphs of text.

When using NLP to extract information and insight from free-form text, the starting point is typically the raw documents stored in object storage such as Azure Storage or Azure Data Lake Store.

Challenges

- Processing a collection of free-form text documents is typically computationally resource intensive, as well as being time intensive.
- Without a standardized document format, it can be difficult to achieve consistently accurate results using free-form text processing to extract specific facts from a document. For example, think of a text representation of an invoice—it can be difficult to build a process that correctly extracts the invoice number and invoice date for invoices across any number of vendors.

What are your options when choosing an NLP service?

In Azure, the following services provide natural language processing (NLP) capabilities:

- Azure HDInsight with Spark and Spark MLlib
- Azure Databricks
- Microsoft Cognitive Services

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want to use prebuilt models? If yes, consider using the APIs offered by Microsoft Cognitive Services.
- Do you need to train custom models against a large corpus of text data? If yes, consider using Azure HDInsight with Spark MLlib and Spark NLP.
- Do you need low-level NLP capabilities like tokenization, stemming, lemmatization, and term frequency/inverse document frequency (TF/IDF)? If yes, consider using Azure HDInsight with Spark MLlib and Spark NLP.
- Do you need simple, high-level NLP capabilities like entity and intent identification, topic detection, spell check, or sentiment analysis? If yes, consider using the APIs offered by Microsoft Cognitive Services.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Provides pretrained models as a service	No	Yes
REST API	Yes	Yes
Programmability	Python, Scala, Java	C#, Java, Node.js, Python, PHP, Ruby
Support processing of big data sets and large documents	Yes	No

Low-level natural language processing capabilities

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Tokenizer	Yes (Spark NLP)	Yes (Linguistic Analysis API)
Stemmer	Yes (Spark NLP)	No
Lemmatizer	Yes (Spark NLP)	No
Part of speech tagging	Yes (Spark NLP)	Yes (Linguistic Analysis API)

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Term frequency/inverse-document frequency (TF/IDF)	Yes (Spark MLlib)	No
String similarity—edit distance calculation	Yes (Spark MLlib)	No
N-gram calculation	Yes (Spark MLlib)	No
Stop word removal	Yes (Spark MLlib)	No

High-level natural language processing capabilities

CAPABILITY	AZURE HDINSIGHT	MICROSOFT COGNITIVE SERVICES
Entity/intent identification and extraction	No	Yes (Language Understanding Intelligent Service (LUIS) API)
Topic detection	Yes (Spark NLP)	Yes (Text Analytics API)
Spell checking	Yes (Spark NLP)	Yes (Bing Spell Check API)
Sentiment analysis	Yes (Spark NLP)	Yes (Text Analytics API)
Language detection	No	Yes (Text Analytics API)
Supports multiple languages besides English	No	Yes (varies by API)

See also

Natural language processing

R developer's guide to Azure

11/19/2021 • 6 minutes to read • Edit Online

Many data scientists dealing with ever-increasing volumes of data are looking for ways to harness the power of cloud computing for their analyses. This article provides an overview of the various ways that data scientists can use their existing skills with the R programming language in Azure.

Microsoft has fully embraced the R programming language as a first-class tool for data scientists. By providing many different options for R developers to run their code in Azure, the company is enabling data scientists to extend their data science workloads into the cloud when tackling large-scale projects.

Let's examine the various options and the most compelling scenarios for each one.

Azure services with R language support

This article covers the following Azure services that support the R language:

SERVICE	DESCRIPTION
Data Science Virtual Machine	a customized VM to use as a data science workstation or as a custom compute target
ML Services on HDInsight	cluster-based system for running R analyses on large datasets across many nodes
Azure Databricks	collaborative Spark environment that supports R and other languages
Azure Machine Learning	cloud service that you use to train, deploy, automate, and manage machine learning models
Azure Batch	offers a variety options for economically running R code across many nodes in a cluster
Azure SQL Managed Instance	run R and Python scripts inside of the SQL Server database engine

Data Science Virtual Machine

The Data Science Virtual Machine (DSVM) is a customized VM image on Microsoft's Azure cloud platform built specifically for doing data science. It has many popular data science tools, including:

- Microsoft R Open
- RStudio Desktop
- RStudio Server

The DSVM can be provisioned with either Windows or Linux as the operating system. You can use the DSVM in two different ways: as an interactive workstation or as a compute platform for a custom cluster.

As a workstation

If you want to get started with R in the cloud quickly and easily, this is your best bet. The environment will be familiar to anyone who has worked with R on a local workstation. However, instead of using local resources, the R environment runs on a VM in the cloud. If your data is already stored in Azure, this has the added benefit of allowing your R scripts to run "closer to the data." Instead of transferring the data across the Internet, the data can be accessed over Azure's internal network, which provides much faster access times.

The DSVM can be particularly useful to small teams of R developers. Instead of investing in powerful workstations for each developer and requiring team members to synchronize on which versions of the various software packages they will use, each developer can spin up an instance of the DSVM whenever needed.

As a compute platform

In addition to being used as a workstation, the DSVM is also used as an elastically scalable compute platform for R projects. Using the `AzureDSVM` R package, you can programmatically control the creation and deletion of DSVM instances. You can form the instances into a cluster and deploy a distributed analysis to be performed in the cloud. This entire process can be controlled by R code running on your local workstation.

To learn more about the DSVM, see Introduction to Azure Data Science Virtual Machine for Linux and Windows.

ML Services on HDInsight

Microsoft ML Services provide data scientists, statisticians, and R programmers with on-demand access to scalable, distributed methods of analytics on HDInsight. This solution provides the latest capabilities for R-based analytics on datasets of virtually any size, loaded to either Azure Blob or Data Lake storage.

This is an enterprise-grade solution that allows you to scale your R code across a cluster. By using functions in Microsoft's `RevoScaleR` package, your R scripts on HDInsight can run data processing functions in parallel across many nodes in a cluster. This allows R to crunch data on a much larger scale than is possible with single-threaded R running on a workstation.

This ability to scale makes ML Services on HDInsight a great option for R developers with massive data sets. It provides a flexible and scalable platform for running your R scripts in the cloud.

For a walk-through on creating an ML Services cluster, see Get started with ML Services on Azure HDInsight.

Azure Databricks

Azure Databricks is an Apache Spark-based analytics platform optimized for the Microsoft Azure cloud services platform. Designed with the founders of Apache Spark, Databricks is integrated with Azure to provide one-click setup, streamlined workflows, and an interactive workspace that enables collaboration between data scientists, data engineers, and business analysts.

The collaboration in Databricks is enabled by the platform's notebook system. Users can create, share, and edit notebooks with other users of the systems. These notebooks allow users to write code that executes against Spark clusters managed in the Databricks environment. These notebooks fully support R and give users access to Spark through both the `SparkR` and `sparklyr` packages.

Since Databricks is built on Spark and has a strong focus on collaboration, the platform is often used by teams of data scientists that work together on complex analyses of large data sets. Because the notebooks in Databricks support other languages in addition to R, it is especially useful for teams where analysts use different languages for their primary work.

The article What is Azure Databricks? can provide more details about the platform and help you get started.

Azure Machine Learning

Azure Machine Learning can be used for any kind of machine learning, from classical machine learning to deep

learning, supervised and unsupervised learning. Whether you prefer to write Python or R code or zero-code/low-code options such as the designer, you can build, train and track highly accurate machine learning and deep-learning models in an Azure Machine Learning Workspace.

Start training on your local machine and then scale out to the cloud. Train your first model in R with Azure Machine Learning today.

Azure Batch

For large-scale R jobs, you can use Azure Batch. This service provides cloud-scale job scheduling and compute management so you can scale your R workload across tens, hundreds, or thousands of virtual machines. Since it is a generalized computing platform, there a few options for running R jobs on Azure Batch.

One option for running an R script in Azure Batch is to bundle your code with "RScript.exe" as a Batch App in the Azure portal. For a detailed walkthrough, see R Workloads on Azure Batch.

Another option is to use the Azure Distributed Data Engineering Toolkit (AZTK), which allows you to provision on-demand Spark clusters using Docker containers in Azure Batch. This provides an economical way to run Spark jobs in Azure. By using SparklyR with AZTK, your R scripts can be scaled out in the cloud easily and economically.

Azure SQL Managed Instance

Azure SQL Managed Instance is Microsoft's intelligent, scalable, cloud database service. It allows you to use the full power of SQL Server without any hassle of setting up the infrastructure. This includes Machine Learning Services which contains Microsoft R and Python packages for high-performance predictive analytics and machine learning.

Machine Learning Services offers an embedded, predictive analytics and data science engine that can execute R/Python code within a SQL Server database. Instead of extracting data from the database and loading it into the R/Python environment, you load your R/Python code directly into the database and let it run right alongside the data. The relational data can be used in stored procedures, as T-SQL scripts containing R/Python statements, or as R/Python code containing T-SQL.

While Machine Learning Services has been part of on-premises SQL Server since 2016, it is relatively new to Azure SQL Managed Instance.

Next steps

- Running your R code on Azure with mrsdeploy
- R on Azure: an overview of packages, tools, and case studies for using R with Azure

The R logo is © 2016 The R Foundation and is used under the terms of the Creative Commons Attribution-ShareAlike 4.0 International license.

What is the Team Data Science Process?

11/19/2021 • 4 minutes to read • Edit Online

The Team Data Science Process (TDSP) is an agile, iterative data science methodology to deliver predictive analytics solutions and intelligent applications efficiently. TDSP helps improve team collaboration and learning by suggesting how team roles work best together. TDSP includes best practices and structures from Microsoft and other industry leaders to help toward successful implementation of data science initiatives. The goal is to help companies fully realize the benefits of their analytics program.

This article provides an overview of TDSP and its main components. We provide a generic description of the process here that can be implemented with different kinds of tools. A more detailed description of the project tasks and roles involved in the lifecycle of the process is provided in additional linked topics. Guidance on how to implement the TDSP using a specific set of Microsoft tools and infrastructure that we use to implement the TDSP in our teams is also provided.

Key components of the TDSP

TDSP has the following key components:

- A **data science lifecycle** definition
- A **standardized project structure**
- **Infrastructure and resources** recommended for data science projects
- **Tools and utilities** recommended for project execution

Data science lifecycle

The Team Data Science Process (TDSP) provides a lifecycle to structure the development of your data science projects. The lifecycle outlines the full steps that successful projects follow.

If you are using another data science lifecycle, such as CRISP-DM, KDD, or your organization's own custom process, you can still use the task-based TDSP in the context of those development lifecycles. At a high level, these different methodologies have much in common.

This lifecycle has been designed for data science projects that ship as part of intelligent applications. These applications deploy machine learning or artificial intelligence models for predictive analytics. Exploratory data science projects or improvised analytics projects can also benefit from using this process. But in such cases some of the steps described may not be needed.

The lifecycle outlines the major stages that projects typically execute, often iteratively:

- **Business Understanding**
- **Data Acquisition and Understanding**
- **Modeling**
- **Deployment**

Here is a visual representation of the **Team Data Science Process lifecycle**.

The goals, tasks, and documentation artifacts for each stage of the lifecycle in TDSP are described in the Team Data Science Process lifecycle topic. These tasks and artifacts are associated with project roles:

- Solution architect
- Project manager
- Data engineer
- Data scientist
- Application developer
- Project lead

The following diagram provides a grid view of the tasks (in blue) and artifacts (in green) associated with each stage of the lifecycle (on the horizontal axis) for these roles (on the vertical axis).

Standardized project structure

Having all projects share a directory structure and use templates for project documents makes it easy for the team members to find information about their projects. All code and documents are stored in a version control system (VCS) like Git, TFS, or Subversion to enable team collaboration. Tracking tasks and features in an agile project tracking system like Jira, Rally, and Azure DevOps allows closer tracking of the code for individual features. Such tracking also enables teams to obtain better cost estimates. TDSP recommends creating a separate repository for each project on the VCS for versioning, information security, and collaboration. The standardized structure for all projects helps build institutional knowledge across the organization.

We provide templates for the folder structure and required documents in standard locations. This folder structure organizes the files that contain code for data exploration and feature extraction, and that record model iterations. These templates make it easier for team members to understand work done by others and to add new members to teams. It is easy to view and update document templates in markdown format. Use templates to provide checklists with key questions for each project to insure that the problem is well defined and that deliverables meet the quality expected. Examples include:

- a project charter to document the business problem and scope of the project
- data reports to document the structure and statistics of the raw data
- model reports to document the derived features
- model performance metrics such as ROC curves or MSE

The directory structure can be cloned from GitHub.

Infrastructure and resources for data science projects

TDSP provides recommendations for managing shared analytics and storage infrastructure such as:

- cloud file systems for storing datasets
- databases
- big data (SQL or Spark) clusters
- machine learning service

The analytics and storage infrastructure, where raw and processed datasets are stored, may be in the cloud or on-premises. This infrastructure enables reproducible analysis. It also avoids duplication, which may lead to inconsistencies and unnecessary infrastructure costs. Tools are provided to provision the shared resources, track them, and allow each team member to connect to those resources securely. It is also a good practice to have project members create a consistent compute environment. Different team members can then replicate and validate experiments.

Here is an example of a team working on multiple projects and sharing various cloud analytics infrastructure components.

Tools and utilities for project execution

Introducing processes in most organizations is challenging. Tools provided to implement the data science process and lifecycle help lower the barriers to and increase the consistency of their adoption. TDSP provides an initial set of tools and scripts to jump-start adoption of TDSP within a team. It also helps automate some of the common tasks in the data science lifecycle such as data exploration and baseline modeling. There is a well-defined structure provided for individuals to contribute shared tools and utilities into their team's shared code repository. These resources can then be leveraged by other projects within the team or the organization. Microsoft provides extensive tooling inside Azure Machine Learning supporting both open-source (Python, R, ONNX, and common deep-learning frameworks) and also Microsoft's own tooling (AutoML).

Next steps

Team Data Science Process: Roles and tasks Outlines the key personnel roles and their associated tasks for a data science team that standardizes on this process.

The Team Data Science Process lifecycle

11/19/2021 • 2 minutes to read • Edit Online

The Team Data Science Process (TDSP) provides a recommended lifecycle that you can use to structure your data-science projects. The lifecycle outlines the complete steps that successful projects follow. If you use another data-science lifecycle, such as the Cross Industry Standard Process for Data Mining (CRISP-DM), Knowledge Discovery in Databases (KDD), or your organization's own custom process, you can still use the task-based TDSP.

This lifecycle is designed for data-science projects that are intended to ship as part of intelligent applications. These applications deploy machine learning or artificial intelligence models for predictive analytics. Exploratory data-science projects and improvised analytics projects can also benefit from the use of this process. But for those projects, some of the steps described here might not be needed.

Five lifecycle stages

The TDSP lifecycle is composed of five major stages that are executed iteratively. These stages include:

1. Business understanding
2. Data acquisition and understanding
3. Modeling
4. Deployment
5. Customer acceptance

Here is a visual representation of the TDSP lifecycle:

The TDSP lifecycle is modeled as a sequence of iterated steps that provide guidance on the tasks needed to use

predictive models. You deploy the predictive models in the production environment that you plan to use to build the intelligent applications. The goal of this process lifecycle is to continue to move a data-science project toward a clear engagement end point. Data science is an exercise in research and discovery. The ability to communicate tasks to your team and your customers by using a well-defined set of artifacts that employ standardized templates helps to avoid misunderstandings. Using these templates also increases the chance of the successful completion of a complex data-science project.

For each stage, we provide the following information:

- **Goals**: The specific objectives.
- **How to do it**: An outline of the specific tasks and guidance on how to complete them.
- **Artifacts**: The deliverables and the support to produce them.

Next steps

We provide full end-to-end walkthroughs that demonstrate all the steps in the process for specific scenarios. The Example walkthroughs article provides a list of the scenarios with links and thumbnail descriptions. The walkthroughs illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

For examples of how to execute steps in TDSPs that use Azure Machine Learning Studio, see Use the TDSP with Azure Machine Learning.

The business understanding stage of the Team Data Science Process lifecycle

11/19/2021 • 3 minutes to read • Edit Online

This article outlines the goals, tasks, and deliverables associated with the business understanding stage of the Team Data Science Process (TDSP). This process provides a recommended lifecycle that you can use to structure your data-science projects. The lifecycle outlines the major stages that projects typically execute, often iteratively:

1. **Business understanding**
2. **Data acquisition and understanding**
3. **Modeling**
4. **Deployment**
5. **Customer acceptance**

Here is a visual representation of the TDSP lifecycle:

Goals

- Specify the key variables that are to serve as the model targets and whose related metrics are used determine the success of the project.
- Identify the relevant data sources that the business has access to or needs to obtain.

How to do it

There are two main tasks addressed in this stage:

- **Define objectives**: Work with your customer and other stakeholders to understand and identify the business problems. Formulate questions that define the business goals that the data science techniques can target.
- **Identify data sources**: Find the relevant data that helps you answer the questions that define the objectives of the project.

Define objectives

1. A central objective of this step is to identify the key business variables that the analysis needs to predict. We refer to these variables as the *model targets*, and we use the metrics associated with them to determine the success of the project. Two examples of such targets are sales forecasts or the probability of an order being fraudulent.

2. Define the project goals by asking and refining "sharp" questions that are relevant, specific, and unambiguous. Data science is a process that uses names and numbers to answer such questions. You typically use data science or machine learning to answer five types of questions:

   - How much or how many? (regression)
   - Which category? (classification)
   - Which group? (clustering)
   - Is this weird? (anomaly detection)
   - Which option should be taken? (recommendation)

   Determine which of these questions you're asking and how answering it achieves your business goals.

3. Define the project team by specifying the roles and responsibilities of its members. Develop a high-level milestone plan that you iterate on as you discover more information.

4. Define the success metrics. For example, you might want to achieve a customer churn prediction. You need an accuracy rate of "x" percent by the end of this three-month project. With this data, you can offer customer promotions to reduce churn. The metrics must be **SMART**:

   - **S**pecific
   - **M**easurable
   - **A**chievable
   - **R**elevant
   - **T**ime-bound

Identify data sources

Identify data sources that contain known examples of answers to your sharp questions. Look for the following data:

- Data that's relevant to the question. Do you have measures of the target and features that are related to the target?
- Data that's an accurate measure of your model target and the features of interest.

For example, you might find that the existing systems need to collect and log additional kinds of data to address the problem and achieve the project goals. In this situation, you might want to look for external data sources or update your systems to collect new data.

Artifacts

Here are the deliverables in this stage:

- Charter document: A standard template is provided in the TDSP project structure definition. The charter

document is a living document. You update the template throughout the project as you make new discoveries and as business requirements change. The key is to iterate upon this document, adding more detail, as you progress through the discovery process. Keep the customer and other stakeholders involved in making the changes and clearly communicate the reasons for the changes to them.

- Data sources: The **Raw data sources** section of the **Data definitions** report that's found in the TDSP project **Data report** folder contains the data sources. This section specifies the original and destination locations for the raw data. In later stages, you fill in additional details like the scripts to move the data to your analytic environment.
- Data dictionaries: This document provides descriptions of the data that's provided by the client. These descriptions include information about the schema (the data types and information on the validation rules, if any) and the entity-relation diagrams, if available.

Next steps

Here are links to each step in the lifecycle of the TDSP:

1. Business understanding
2. Data acquisition and understanding
3. Modeling
4. Deployment
5. Customer acceptance

We provide full walkthroughs that demonstrate all the steps in the process for specific scenarios. The Example walkthroughs article provides a list of the scenarios with links and thumbnail descriptions. The walkthroughs illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

Data acquisition and understanding stage of the Team Data Science Process

11/19/2021 • 3 minutes to read • Edit Online

This article outlines the goals, tasks, and deliverables associated with the data acquisition and understanding stage of the Team Data Science Process (TDSP). This process provides a recommended lifecycle that you can use to structure your data-science projects. The lifecycle outlines the major stages that projects typically execute, often iteratively:

1. **Business understanding**
2. **Data acquisition and understanding**
3. **Modeling**
4. **Deployment**
5. **Customer acceptance**

Here is a visual representation of the TDSP lifecycle:

Goals

- Produce a clean, high-quality data set whose relationship to the target variables is understood. Locate the data set in the appropriate analytics environment so you are ready to model.
- Develop a solution architecture of the data pipeline that refreshes and scores the data regularly.

How to do it

There are three main tasks addressed in this stage:

- **Ingest the data** into the target analytic environment.
- **Explore the data** to determine if the data quality is adequate to answer the question.
- **Set up a data pipeline** to score new or regularly refreshed data.

Ingest the data

Set up the process to move the data from the source locations to the target locations where you run analytics operations, like training and predictions. For technical details and options on how to move the data with various Azure data services, see Load data into storage environments for analytics.

Explore the data

Before you train your models, you need to develop a sound understanding of the data. Real-world data sets are often noisy, are missing values, or have a host of other discrepancies. You can use data summarization and visualization to audit the quality of your data and provide the information you need to process the data before it's ready for modeling. This process is often iterative. For guidance on cleaning the data, see Tasks to prepare data for enhanced machine learning.

After you're satisfied with the quality of the cleansed data, the next step is to better understand the patterns that are inherent in the data. This data analysis helps you choose and develop an appropriate predictive model for your target. Look for evidence for how well connected the data is to the target. Then determine whether there is sufficient data to move forward with the next modeling steps. Again, this process is often iterative. You might need to find new data sources with more accurate or more relevant data to augment the data set initially identified in the previous stage.

Set up a data pipeline

In addition to the initial ingestion and cleaning of the data, you typically need to set up a process to score new data or refresh the data regularly as part of an ongoing learning process. Scoring may be completed with a data pipeline or workflow. The Move data from a SQL Server instance to Azure SQL Database with Azure Data Factory article gives an example of how to set up a pipeline with Azure Data Factory.

In this stage, you develop a solution architecture of the data pipeline. You develop the pipeline in parallel with the next stage of the data science project. Depending on your business needs and the constraints of your existing systems into which this solution is being integrated, the pipeline can be one of the following options:

- Batch-based
- Streaming or real time
- A hybrid

Artifacts

The following are the deliverables in this stage:

- Data quality report: This report includes data summaries, the relationships between each attribute and target, variable ranking, and more.
- **Solution architecture**: The solution architecture can be a diagram or description of your data pipeline that you use to run scoring or predictions on new data after you have built a model. It also contains the pipeline to retrain your model based on new data. Store the document in the Project directory when you use the TDSP directory structure template.
- **Checkpoint decision**: Before you begin full-feature engineering and model building, you can reevaluate the project to determine whether the value expected is sufficient to continue pursuing it. You might, for example, be ready to proceed, need to collect more data, or abandon the project as the data does not exist to answer the question.

Next steps

Here are links to each step in the lifecycle of the TDSP:

1. Business understanding
2. Data acquisition and understanding
3. Modeling
4. Deployment
5. Customer acceptance

We provide full walkthroughs that demonstrate all the steps in the process for specific scenarios. The Example walkthroughs article provides a list of the scenarios with links and thumbnail descriptions. The walkthroughs illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

For examples of how to execute steps in TDSPs that use Azure Machine Learning Studio, see Use the TDSP with Azure Machine Learning.

Modeling stage of the Team Data Science Process lifecycle

11/19/2021 • 3 minutes to read • Edit Online

This article outlines the goals, tasks, and deliverables associated with the modeling stage of the Team Data Science Process (TDSP). This process provides a recommended lifecycle that you can use to structure your data-science projects. The lifecycle outlines the major stages that projects typically execute, often iteratively:

1. **Business understanding**
2. **Data acquisition and understanding**
3. **Modeling**
4. **Deployment**
5. **Customer acceptance**

Here is a visual representation of the TDSP lifecycle:

Goals

- Determine the optimal data features for the machine-learning model.
- Create an informative machine-learning model that predicts the target most accurately.
- Create a machine-learning model that's suitable for production.

How to do it

There are three main tasks addressed in this stage:

- **Feature engineering**: Create data features from the raw data to facilitate model training.
- **Model training**: Find the model that answers the question most accurately by comparing their success metrics.
- Determine if your model is **suitable for production.**

Feature engineering

Feature engineering involves the inclusion, aggregation, and transformation of raw variables to create the features used in the analysis. If you want insight into what is driving a model, then you need to understand how the features relate to each other and how the machine-learning algorithms are to use those features.

This step requires a creative combination of domain expertise and the insights obtained from the data exploration step. Feature engineering is a balancing act of finding and including informative variables, but at the same time trying to avoid too many unrelated variables. Informative variables improve your result; unrelated variables introduce unnecessary noise into the model. You also need to generate these features for any new data obtained during scoring. As a result, the generation of these features can only depend on data that's available at the time of scoring.

For technical guidance on feature engineering when make use of various Azure data technologies, see Feature engineering in the data science process.

Model training

Depending on the type of question that you're trying to answer, there are many modeling algorithms available. For guidance on choosing the algorithms, see How to choose algorithms for Microsoft Azure Machine Learning. Although this article uses Azure Machine Learning, the guidance it provides is useful for any machine-learning projects.

The process for model training includes the following steps:

- **Split the input data** randomly for modeling into a training data set and a test data set.
- **Build the models** by using the training data set.
- **Evaluate** the training and the test data set. Use a series of competing machine-learning algorithms along with the various associated tuning parameters (known as a *parameter sweep*) that are geared toward answering the question of interest with the current data.
- **Determine the "best" solution** to answer the question by comparing the success metrics between alternative methods.

> **NOTE**
>
> **Avoid leakage**: You can cause data leakage if you include data from outside the training data set that allows a model or machine-learning algorithm to make unrealistically good predictions. Leakage is a common reason why data scientists get nervous when they get predictive results that seem too good to be true. These dependencies can be hard to detect. To avoid leakage often requires iterating between building an analysis data set, creating a model, and evaluating the accuracy of the results.

Artifacts

The artifacts produced in this stage include:

- Feature sets: The features developed for the modeling are described in the **Feature sets** section of the **Data definition** report. It contains pointers to the code to generate the features and a description of how the feature was generated.
- Model report: For each model that's tried, a standard, template-based report that provides details on each

experiment is produced.

- **Checkpoint decision**: Evaluate whether the model performs sufficiently for production. Some key questions to ask are:
  - Does the model answer the question with sufficient confidence given the test data?
  - Should you try any alternative approaches? Should you collect additional data, do more feature engineering, or experiment with other algorithms?

Next steps

Here are links to each step in the lifecycle of the TDSP:

1. Business understanding
2. Data acquisition and understanding
3. Modeling
4. Deployment
5. Customer acceptance

We provide full end-to-end walkthroughs that demonstrate all the steps in the process for specific scenarios. The Example walkthroughs article provides a list of the scenarios with links and thumbnail descriptions. The walkthroughs illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

For examples of how to execute steps in TDSPs that use Azure Machine Learning Studio, see Use the TDSP with Azure Machine Learning.

Deployment stage of the Team Data Science Process lifecycle

11/19/2021 • 2 minutes to read • Edit Online

This article outlines the goals, tasks, and deliverables associated with the deployment of the Team Data Science Process (TDSP). This process provides a recommended lifecycle that you can use to structure your data-science projects. The lifecycle outlines the major stages that projects typically execute, often iteratively:

1. **Business understanding**
2. **Data acquisition and understanding**
3. **Modeling**
4. **Deployment**
5. **Customer acceptance**

Here is a visual representation of the TDSP lifecycle:

Goal

Deploy models with a data pipeline to a production or production-like environment for final user acceptance.

How to do it

The main task addressed in this stage:

Operationalize the model: Deploy the model and pipeline to a production or production-like environment for

application consumption.

Operationalize a model

After you have a set of models that perform well, you can operationalize them for other applications to consume. Depending on the business requirements, predictions are made either in real time or on a batch basis. To deploy models, you expose them with an open API interface. The interface enables the model to be easily consumed from various applications, such as:

- Online websites
- Spreadsheets
- Dashboards
- Line-of-business applications
- Back-end applications

For examples of model operationalization with an Machine Learning Studio (classic) web service, see Deploy an Machine Learning Studio (classic) web service. It is a best practice to build telemetry and monitoring into the production model and the data pipeline that you deploy. This practice helps with subsequent system status reporting and troubleshooting.

Artifacts

- A status dashboard that displays the system health and key metrics
- A final modeling report with deployment details
- A final solution architecture document

Next steps

Here are links to each step in the lifecycle of the TDSP:

1. Business understanding
2. Data Acquisition and understanding
3. Modeling
4. Deployment
5. Customer acceptance

We provide full walkthroughs that demonstrate all the steps in the process for specific scenarios. The Example walkthroughs article provides a list of the scenarios with links and thumbnail descriptions. The walkthroughs illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

For examples of how to execute steps in TDSPs that use Machine Learning Studio (classic), see Use the TDSP with Azure Machine Learning.

Customer acceptance stage of the Team Data Science Process lifecycle

11/19/2021 • 2 minutes to read • Edit Online

This article outlines the goals, tasks, and deliverables associated with the customer acceptance stage of the Team Data Science Process (TDSP). This process provides a recommended lifecycle that you can use to structure your data-science projects. The lifecycle outlines the major stages that projects typically execute, often iteratively:

1. **Business understanding**
2. **Data acquisition and understanding**
3. **Modeling**
4. **Deployment**
5. **Customer acceptance**

Here is a visual representation of the TDSP lifecycle:

Goal

Finalize project deliverables: Confirm that the pipeline, the model, and their deployment in a production environment satisfy the customer's objectives.

How to do it

There are two main tasks addressed in this stage:

- **System validation**: Confirm that the deployed model and pipeline meet the customer's needs.
- **Project hand-off**: Hand the project off to the entity that's going to run the system in production.

The customer should validate that the system meets their business needs and that it answers the questions with acceptable accuracy to deploy the system to production for use by their client's application. All the documentation is finalized and reviewed. The project is handed-off to the entity responsible for operations. This entity might be, for example, an IT or customer data-science team or an agent of the customer that's responsible for running the system in production.

Artifacts

The main artifact produced in this final stage is the **Exit report of the project for the customer**. This technical report contains all the details of the project that are useful for learning about how to operate the system. TDSP provides an Exit report template. You can use the template as is, or you can customize it for specific client needs.

Next steps

Here are links to each step in the lifecycle of the TDSP:

1. Business understanding
2. Data acquisition and understanding
3. Modeling
4. Deployment
5. Customer acceptance

We provide full walkthroughs that demonstrate all the steps in the process for specific scenarios. The Example walkthroughs article provides a list of the scenarios with links and thumbnail descriptions. The walkthroughs illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

For examples of how to execute steps in TDSPs that use Azure Machine Learning Studio, see Use the TDSP with Azure Machine Learning.

Team Data Science Process roles and tasks

11/19/2021 • 6 minutes to read • Edit Online

The Team Data Science Process (TDSP) is a framework developed by Microsoft that provides a structured methodology to efficiently build predictive analytics solutions and intelligent applications. This article outlines the key personnel roles and associated tasks for a data science team standardizing on this process.

This introductory article links to tutorials on how to set up the TDSP environment. The tutorials provide detailed guidance for using Azure DevOps Projects, Azure Repos repositories, and Azure Boards. The motivating goal is moving from concept through modeling and into deployment.

The tutorials use Azure DevOps because that is how to implement TDSP at Microsoft. Azure DevOps facilitates collaboration by integrating role-based security, work item management and tracking, and code hosting, sharing, and source control. The tutorials also use an Azure Data Science Virtual Machine (DSVM) as the analytics desktop, which has several popular data science tools pre-configured and integrated with Microsoft software and Azure services.

You can use the tutorials to implement TDSP using other code-hosting, agile planning, and development tools and environments, but some features may not be available.

Structure of data science groups and teams

Data science functions in enterprises are often organized in the following hierarchy:

- Data science group
  - Data science team/s within the group

In such a structure, there are group leads and team leads. Typically, a data science project is done by a data science team. Data science teams have project leads for project management and governance tasks, and individual data scientists and engineers to perform the data science and data engineering parts of the project. The initial project setup and governance is done by the group, team, or project leads.

Definition and tasks for the four TDSP roles

With the assumption that the data science unit consists of teams within a group, there are four distinct roles for TDSP personnel:

1. **Group Manager**: Manages the entire data science unit in an enterprise. A data science unit might have multiple teams, each of which is working on multiple data science projects in distinct business verticals. A Group Manager might delegate their tasks to a surrogate, but the tasks associated with the role do not change.

2. **Team Lead**: Manages a team in the data science unit of an enterprise. A team consists of multiple data scientists. For a small data science unit, the Group Manager and the Team Lead might be the same person.

3. **Project Lead**: Manages the daily activities of individual data scientists on a specific data science project.

4. **Project Individual Contributors**: Data Scientists, Business Analysts, Data Engineers, Architects, and others who execute a data science project.

> **NOTE**
> Depending on the structure and size of an enterprise, a single person may play more than one role, or more than one person may fill a role.

Tasks to be completed by the four roles

The following diagram shows the top-level tasks for each Team Data Science Process role. This schema and the following, more detailed outline of tasks for each TDSP role can help you choose the tutorial you need based on your responsibilities.

Group Manager tasks

The Group Manager or a designated TDSP system administrator completes the following tasks to adopt the TDSP:

- Creates an Azure DevOps **organization** and a group project within the organization.
- Creates a **project template repository** in the Azure DevOps group project, and seeds it from the project template repository developed by the Microsoft TDSP team. The Microsoft TDSP project template repository provides:
  - A **standardized directory structure**, including directories for data, code, and documents.
  - A set of **standardized document templates** to guide an efficient data science process.
- Creates a **utility repository**, and seeds it from the utility repository developed by the Microsoft TDSP team. The TDSP utility repository from Microsoft provides a set of useful utilities to make the work of a data scientist more efficient. The Microsoft utility repository includes utilities for interactive data exploration, analysis, reporting, and baseline modeling and reporting.
- Sets up the **security control policy** for the organization account.

For detailed instructions, see Group Manager tasks for a data science team.

Team Lead tasks

The Team Lead or a designated project administrator completes the following tasks to adopt the TDSP:

- Creates a team **project** in the group's Azure DevOps organization.
- Creates the **project template repository** in the project, and seeds it from the group project template

repository set up by the Group Manager or delegate.

- Creates the **team utility repository**, seeds it from the group utility repository, and adds team-specific utilities to the repository.
- Optionally creates Azure file storage to store useful data assets for the team. Other team members can mount this shared cloud file store on their analytics desktops.
- Optionally mounts the Azure file storage on the team's **DSVM** and adds team data assets to it.
- Sets up **security control** by adding team members and configuring their permissions.

For detailed instructions, see Team Lead tasks for a data science team.

Project Lead tasks

The Project Lead completes the following tasks to adopt the TDSP:

- Creates a **project repository** in the team project, and seeds it from the project template repository.
- Optionally creates **Azure file storage** to store the project's data assets.
- Optionally mounts the Azure file storage to the **DSVM** and adds project data assets to it.
- Sets up **security control** by adding project members and configuring their permissions.

For detailed instructions, see Project Lead tasks for a data science team.

Project Individual Contributor tasks

The Project Individual Contributor, usually a Data Scientist, conducts the following tasks using the TDSP:

- Clones the **project repository** set up by the project lead.
- Optionally mounts the shared team and project **Azure file storage** on their **Data Science Virtual Machine** (DSVM).
- Executes the project.

For detailed instructions for onboarding onto a project, see Project Individual Contributor tasks for a data science team.

Data science project execution workflow

By following the relevant tutorials, data scientists, project leads, and team leads can create work items to track all tasks and stages for project from beginning to end. Using Azure Repos promotes collaboration among data scientists and ensures that the artifacts generated during project execution are version controlled and shared by all project members. Azure DevOps lets you link your Azure Boards work items with your Azure Repos repository branches and easily track what has been done for a work item.

The following figure outlines the TDSP workflow for project execution:

The workflow steps can be grouped into three activities:

- Project Leads conduct sprint planning
- Data Scientists develop artifacts on `git` branches to address work items
- Project Leads or other team members do code reviews and merge working branches to the primary branch

For detailed instructions on project execution workflow, see Agile development of data science projects.

TDSP project template repository

Use the Microsoft TDSP team's project template repository to support efficient project execution and collaboration. The repository gives you a standardized directory structure and document templates you can use for your own TDSP projects.

Next steps

Explore more detailed descriptions of the roles and tasks defined by the Team Data Science Process:

- Group Manager tasks for a data science team
- Team Lead tasks for a data science team
- Project Lead tasks for a data science team
- Project Individual Contributor tasks for a data science team

Team Data Science Process group manager tasks

11/19/2021 • 7 minutes to read • Edit Online

This article describes the tasks that a *group manager* completes for a data science organization. The group manager manages the entire data science unit in an enterprise. A data science unit may have several teams, each of which is working on many data science projects in distinct business verticals. The group manager's objective is to establish a collaborative group environment that standardizes on the Team Data Science Process (TDSP). For an outline of all the personnel roles and associated tasks handled by a data science team standardizing on the TDSP, see Team Data Science Process roles and tasks.

The following diagram shows the six main group manager setup tasks. Group managers may delegate their tasks to surrogates, but the tasks associated with the role don't change.

1. Set up an **Azure DevOps organization** for the group.
2. Create the default **GroupCommon project** in the Azure DevOps organization.
3. Create the **GroupProjectTemplate** repository in Azure Repos.
4. Create the **GroupUtilities** repository in Azure Repos.
5. Import the contents of the Microsoft TDSP team's **ProjectTemplate** and **Utilities** repositories into the group common repositories.
6. Set up **membership** and **permissions** for team members to access the group.

The following tutorial walks through the steps in detail.

> **NOTE**
>
> This article uses Azure DevOps to set up a TDSP group environment, because that is how to implement TDSP at Microsoft. If your group uses other code hosting or development platforms, the Group Manager's tasks are the same, but the way to complete them may be different.

Create an organization and project in Azure DevOps

1. Go to visualstudio.microsoft.com, select **Sign in** at upper right, and sign into your Microsoft account.

If you don't have a Microsoft account, select **Sign up now**, create a Microsoft account, and sign in using this account. If your organization has a Visual Studio subscription, sign in with the credentials for that

subscription.

2. After you sign in, at upper right on the Azure DevOps page, select **Create new organization**.

3. If you're prompted to agree to the Terms of Service, Privacy Statement, and Code of Conduct, select **Continue**.

4. In the signup dialog, name your Azure DevOps organization and accept the host region assignment, or drop down and select a different region. Then select **Continue**.

5. Under **Create a project to get started**, enter *GroupCommon*, and then select **Create project**.

The **GroupCommon** project **Summary** page opens. The page URL is *https://<servername>/<organization-name>/GroupCommon*.

Set up the group common repositories

Azure Repos hosts the following types of repositories for your group:

- **Group common repositories**: General-purpose repositories that multiple teams within a data science unit can adopt for many data science projects.
- **Team repositories**: Repositories for specific teams within a data science unit. These repositories are specific for a team's needs, and may be used for multiple projects within that team, but are not general enough to be used across multiple teams within a data science unit.
- **Project repositories**: Repositories for specific projects. Such repositories may not be general enough for multiple projects within a team, or for other teams in a data science unit.

To set up the group common repositories in your project, you:

- Rename the default **GroupCommon** repository to **GroupProjectTemplate**
- Create a new **GroupUtilities** repository

Rename the default project repository to GroupProjectTemplate

To rename the default **GroupCommon** project repository to **GroupProjectTemplate**:

1. On the **GroupCommon** project **Summary** page, select **Repos**. This action takes you to the default **GroupCommon** repository of the GroupCommon project, which is currently empty.

2. At the top of the page, drop down the arrow next to **GroupCommon** and select **Manage repositories**.

3. On the **Project Settings** page, select the **...** next to **GroupCommon**, and then select **Rename repository**.

4. In the **Rename the GroupCommon repository** popup, enter *GroupProjectTemplate*, and then select **Rename**.

Create the GroupUtilities repository

To create the **GroupUtilities** repository:

1. On the **GroupCommon** project **Summary** page, select **Repos**.

2. At the top of the page, drop down the arrow next to **GroupProjectTemplate** and select **New repository**.

3. In the **Create a new repository** dialog, select **Git** as the **Type**, enter *GroupUtilities* as the **Repository name**, and then select **Create**.

4. On the **Project Settings** page, select **Repositories** under **Repos** in the left navigation to see the two group repositories: **GroupProjectTemplate** and **GroupUtilities**.

Import the Microsoft TDSP team repositories

In this part of the tutorial, you import the contents of the **ProjectTemplate** and **Utilities** repositories managed by the Microsoft TDSP team into your **GroupProjectTemplate** and **GroupUtilities** repositories.

To import the TDSP team repositories:

1. From the **GroupCommon** project home page, select **Repos** in the left navigation. The default **GroupProjectTemplate** repo opens.

2. On the **GroupProjectTemplate is empty** page, select **Import**.

3. In the **Import a Git repository** dialog, select **Git** as the **Source type**, and enter *https://github.com/Azure/Azure-TDSP-ProjectTemplate.git* for the **Clone URL**. Then select **Import**. The contents of the Microsoft TDSP team ProjectTemplate repository are imported into your GroupProjectTemplate repository.

4. At the top of the **Repos** page, drop down and select the **GroupUtilities** repository.

Each of your two group repositories now contains all the files, except those in the *.git* directory, from the Microsoft TDSP team's corresponding repository.

Customize the contents of the group repositories

If you want to customize the contents of your group repositories to meet the specific needs of your group, you can do that now. You can modify the files, change the directory structure, or add files that your group has developed or that are helpful for your group.

Make changes in Azure Repos

To customize repository contents:

1. On the **GroupCommon** project **Summary** page, select **Repos**.
2. At the top of the page, select the repository you want to customize.
3. In the repo directory structure, navigate to the folder or file you want to change.

- To create new folders or files, select the arrow next to **New**.

- To upload files, select **Upload file(s)**.

- To edit existing files, navigate to the file and then select **Edit**.

4. After adding or editing files, select **Commit**.

Make changes using your local machine or DSVM

If you want to make changes using your local machine or DSVM and push the changes up to the group repositories, make sure you have the prerequisites for working with Git and DSVMs:

- An Azure subscription, if you want to create a DSVM.
- Git installed on your machine. If you're using a DSVM, Git is pre-installed. Otherwise, see the Platforms and tools appendix.
- If you want to use a DSVM, the Windows or Linux DSVM created and configured in Azure. For more information and instructions, see the Data Science Virtual Machine Documentation.
- For a Windows DSVM, Git Credential Manager (GCM) installed on your machine. In the *README.md* file, scroll down to the **Download and Install** section and select the **latest installer**. Download the *.exe* installer from the installer page and run it.

- For a Linux DSVM, an SSH public key set up on your DSVM and added in Azure DevOps. For more information and instructions, see the **Create SSH public key** section in the Platforms and tools appendix.

First, copy or *clone* the repository to your local machine.

1. On the **GroupCommon** project **Summary** page, select **Repos**, and at the top of the page, select the repository you want to clone.
2. On the repo page, select **Clone** at upper right.
3. In the **Clone repository** dialog, select **HTTPS** for an HTTP connection, or **SSH** for an SSH connection, and copy the clone URL under **Command line** to your clipboard.

4. On your local machine, create the following directories:
   - For Windows: **C:\GitRepos\GroupCommon**
   - For Linux, **$/GitRepos/GroupCommon** on your home directory
5. Change to the directory you created.
6. In Git Bash, run the command `git clone <clone URL>.`

   For example, either of the following commands clones the **GroupUtilities** repository to the *GroupCommon* directory on your local machine.

   HTTPS connection:

   ```
   git clone https://DataScienceUnit@dev.azure.com/DataScienceUnit/GroupCommon/_git/GroupUtilities
   ```

   SSH connection:

   ```
   git clone git@ssh.dev.azure.com:v3/DataScienceUnit/GroupCommon/GroupUtilities
   ```

After making whatever changes you want in the local clone of your repository, you can push the changes to the shared group common repositories.

Run the following Git Bash commands from your local **GroupProjectTemplate** or **GroupUtilities** directory.

```
git add .
git commit -m "push from local"
git push
```

NOTE

If this is the first time you commit to a Git repository, you may need to configure global parameters *user.name* and *user.email* before you run the `git commit` command. Run the following two commands:

```
git config --global user.name <your name>
```

```
git config --global user.email <your email address>
```

If you're committing to several Git repositories, use the same name and email address for all of them. Using the same name and email address is convenient when building Power BI dashboards to track your Git activities in multiple repositories.

Add group members and configure permissions

To add members to the group:

1. In Azure DevOps, from the **GroupCommon** project home page, select **Project settings** from the left navigation.
2. From the **Project Settings** left navigation, select **Teams**, then on the **Teams** page, select the **GroupCommon Team**.

3. On the **Team Profile** page, select **Add**.

4. In the **Add users and groups** dialog, search for and select members to add to the group, and then select **Save changes**.

To configure permissions for members:

1. From the **Project Settings** left navigation, select **Permissions**.
2. On the **Permissions** page, select the group you want to add members to.
3. On the page for that group, select **Members**, and then select **Add**.
4. In the **Invite members** popup, search for and select members to add to the group, and then select **Save**.

Next steps

Here are links to detailed descriptions of the other roles and tasks in the Team Data Science Process:

- Team Lead tasks for a data science team
- Project Lead tasks for a data science team
- Project Individual Contributor tasks for a data science team

Tasks for the team lead on a Team Data Science Process team

11/19/2021 • 11 minutes to read • Edit Online

This article describes the tasks that a *team lead* completes for their data science team. The team lead's objective is to establish a collaborative team environment that standardizes on the Team Data Science Process (TDSP). The TDSP is designed to help improve collaboration and team learning.

The TDSP is an agile, iterative data science methodology to efficiently deliver predictive analytics solutions and intelligent applications. The process distills the best practices and structures from Microsoft and the industry. The goal is successful implementation of data science initiatives and fully realizing the benefits of their analytics programs. For an outline of the personnel roles and associated tasks for a data science team standardizing on the TDSP, see Team Data Science Process roles and tasks.

A team lead manages a team consisting of several data scientists in the data science unit of an enterprise. Depending on the data science unit's size and structure, the group manager and the team lead might be the same person, or they could delegate their tasks to surrogates. But the tasks themselves do not change.

The following diagram shows the workflow for the tasks the team lead completes to set up a team environment:

1. Create a **team project** in the group's organization in Azure DevOps.
2. Rename the default team repository to **TeamUtilities**.
3. Create a new **TeamTemplate** repository in the team project.
4. Import the contents of the group's **GroupUtilities** and **GroupProjectTemplate** repositories into the **TeamUtilities** and **TeamTemplate** repositories.
5. Set up **security control** by adding team members and configuring their permissions.
6. If required, create team data and analytics resources:
   - Add team-specific utilities to the **TeamUtilities** repository.
   - Create **Azure file storage** to store data assets that can be useful for the entire team.
   - Mount the Azure file storage to the team lead's **Data Science Virtual Machine** (DSVM) and add data assets to it.

The following tutorial walks through the steps in detail.

> **NOTE**
> This article uses Azure DevOps and a DSVM to set up a TDSP team environment, because that is how to implement TDSP at Microsoft. If your team uses other code hosting or development platforms, the team lead tasks are the same, but the way to complete them may be different.

Prerequisites

This tutorial assumes that the following resources and permissions have been set up by your group manager:

- The Azure DevOps **organization** for your data unit
- **GroupProjectTemplate** and **GroupUtilities** repositories, populated with the contents of the Microsoft TDSP team's **ProjectTemplate** and **Utilities** repositories
- Permissions on your organization account for you to create projects and repositories for your team

To be able to clone repositories and modify their content on your local machine or DSVM, or set up Azure file storage and mount it to your DSVM, you need the following:

- An Azure subscription.
- Git installed on your machine. If you're using a DSVM, Git is pre-installed. Otherwise, see the Platforms and tools appendix.
- If you want to use a DSVM, the Windows or Linux DSVM created and configured in Azure. For more information and instructions, see the Data Science Virtual Machine Documentation.
- For a Windows DSVM, Git Credential Manager (GCM) installed on your machine. In the *README.md* file, scroll down to the **Download and Install** section and select the **latest installer**. Download the *.exe* installer from the installer page and run it.
- For a Linux DSVM, an SSH public key set up on your DSVM and added in Azure DevOps. For more information and instructions, see the **Create SSH public key** section in the Platforms and tools appendix.

Create a team project and repositories

In this section, you create the following resources in your group's Azure DevOps organization:

- The **MyTeam** project in Azure DevOps
- The **TeamTemplate** repository
- The **TeamUtilities** repository

The names specified for the repositories and directories in this tutorial assume that you want to establish a separate project for your own team within your larger data science organization. However, the entire group can choose to work under a single project created by the group manager or organization administrator. Then, all the data science teams create repositories under this single project. This scenario might be valid for:

- A small data science group that doesn't have multiple data science teams.
- A larger data science group with multiple data science teams that nevertheless wants to optimize inter-team collaboration with activities such as group-level sprint planning.

If teams choose to have their team-specific repositories under a single group project, the team leads should create the repositories with names like *<TeamName>Template* and *<TeamName>Utilities*. For instance: *TeamATemplate* and *TeamAUtilities*.

In any case, team leads need to let their team members know which template and utilities repositories to set up and clone. Project leads should follow the project lead tasks for a data science team to create project repositories, whether under separate projects or a single project.

Create the MyTeam project

To create a separate project for your team:

1. In your web browser, go to your group's Azure DevOps organization home page at URL *https://<server name>/<organization name>*, and select **New project**.

2. In the **Create project** dialog, enter your team name, such as *MyTeam*, under **Project name**, and then select **Advanced**.

3. Under **Version control**, select **Git**, and under **Work item process**, select **Agile**. Then select **Create**.

The team project **Summary** page opens, with page URL *https://<server name>/<organization name>/<team name>*.

Rename the MyTeam default repository to TeamUtilities

1. On the **MyTeam** project **Summary** page, under **What service would you like to start with?**, select **Repos**.

2. On the **MyTeam** repo page, select the **MyTeam** repository at the top of the page, and then select **Manage repositories** from the dropdown.

3. On the **Project Settings** page, select the **...** next to the **MyTeam** repository, and then select **Rename repository**.

4. In the **Rename the MyTeam repository** popup, enter *TeamUtilities*, and then select **Rename**.

Create the TeamTemplate repository

1. On the **Project Settings** page, select **New repository.**

Or, select **Repos** from the left navigation of the **MyTeam** project **Summary** page, select a repository at the top of the page, and then select **New repository** from the dropdown.

2. In the **Create a new repository** dialog, make sure **Git** is selected under **Type**. Enter *TeamTemplate* under **Repository name**, and then select **Create**.

3. Confirm that you can see the two repositories **TeamUtilities** and **TeamTemplate** on your project settings page.

Import the contents of the group common repositories

To populate your team repositories with the contents of the group common repositories set up by your group manager:

1. From your **MyTeam** project home page, select **Repos** in the left navigation. If you get a message that the **MyTeam** template is not found, select the link in **Otherwise, navigate to your default TeamTemplate repository**.

   The default **TeamTemplate** repository opens.

2. On the **TeamTemplate is empty** page, select **Import**.

3. In the **Import a Git repository** dialog, select **Git** as the **Source type**, and enter the URL for your group common template repository under **Clone URL**. The URL is *https://<server name>/<organization name>/_git/<repository name>*. For example: *https://dev.azure.com/DataScienceUnit/GroupCommon/_git/GroupProjectTemplate*.

4. Select **Import**. The contents of your group template repository are imported into your team template repository.

5. At the top of your project's **Repos** page, drop down and select the **TeamUtilities** repository.

6. Repeat the import process to import the contents of your group common utilities repository, for example *GroupUtilities*, into your **TeamUtilities** repository.

Each of your two team repositories now contains the files from the corresponding group common repository.

Customize the contents of the team repositories

If you want to customize the contents of your team repositories to meet your team's specific needs, you can do that now. You can modify files, change the directory structure, or add files and folders.

To modify, upload, or create files or folders directly in Azure DevOps:

1. On the **MyTeam** project **Summary** page, select **Repos**.
2. At the top of the page, select the repository you want to customize.
3. In the repo directory structure, navigate to the folder or file you want to change.
   - To create new folders or files, select the arrow next to **New**.

   - To upload files, select **Upload file(s)**.

   - To edit existing files, navigate to the file and then select **Edit**.

4. After adding or editing files, select **Commit**.

To work with repositories on your local machine or DSVM, you first copy or *clone* the repositories to your local machine, and then commit and push your changes up to the shared team repositories,

To clone repositories:

1. On the **MyTeam** project **Summary** page, select **Repos**, and at the top of the page, select the repository you want to clone.

2. On the repo page, select **Clone** at upper right.

3. In the **Clone repository** dialog, under **Command line**, select **HTTPS** for an HTTP connection or **SSH** for an SSH connection, and copy the clone URL to your clipboard.

4. On your local machine, create the following directories:
   - For Windows: **C:\GitRepos\MyTeam**
   - For Linux, **$home/GitRepos/MyTeam**
5. Change to the directory you created.
6. In Git Bash, run the command `git clone <clone URL>`, where <clone URL> is the URL you copied from the **Clone** dialog.

   For example, use one of the following commands to clone the **TeamUtilities** repository to the *MyTeam* directory on your local machine.

   HTTPS connection:

   ```
   git clone https://DataScienceUnit@dev.azure.com/DataScienceUnit/MyTeam/_git/TeamUtilities
   ```

   SSH connection:

   ```
   git clone git@ssh.dev.azure.com:v3/DataScienceUnit/MyTeam/TeamUtilities
   ```

After making whatever changes you want in the local clone of your repository, commit and push the changes to the shared team repositories.

Run the following Git Bash commands from your local **GitRepos\MyTeam\TeamTemplate** or **GitRepos\MyTeam\TeamUtilities** directory.

```
git add .
git commit -m "push from local"
git push
```

> **NOTE**
>
> If this is the first time you commit to a Git repository, you may need to configure global parameters *user.name* and *user.email* before you run the `git commit` command. Run the following two commands:
>
> `git config --global user.name <your name>`
>
> `git config --global user.email <your email address>`
>
> If you're committing to several Git repositories, use the same name and email address for all of them. Using the same name and email address is convenient when building Power BI dashboards to track your Git activities in multiple repositories.

Add team members and configure permissions

To add members to the team:

1. In Azure DevOps, from the **MyTeam** project home page, select **Project settings** from the left navigation.
2. From the **Project Settings** left navigation, select **Teams**, then on the **Teams** page, select the **MyTeam Team**.

3. On the **Team Profile** page, select **Add**.

4. In the **Add users and groups** dialog, search for and select members to add to the group, and then select **Save changes**.

To configure permissions for team members:

1. From the **Project Settings** left navigation, select **Permissions**.
2. On the **Permissions** page, select the group you want to add members to.
3. On the page for that group, select **Members**, and then select **Add**.
4. In the **Invite members** popup, search for and select members to add to the group, and then select **Save**.

Create team data and analytics resources

This step is optional, but sharing data and analytics resources with your entire team has performance and cost benefits. Team members can execute their projects on the shared resources, save on budgets, and collaborate more efficiently. You can create Azure file storage and mount it on your DSVM to share with team members.

For information about sharing other resources with your team, such as Azure HDInsight Spark clusters, see Platforms and tools. That topic provides guidance from a data science perspective on selecting resources that are appropriate for your needs, and links to product pages and other relevant and useful tutorials.

> **NOTE**
> To avoid transmitting data across data centers, which might be slow and costly, make sure that your Azure resource group, storage account, and DSVM are all hosted in the same Azure region.

Create Azure file storage

1. Run the following script to create Azure file storage for data assets that are useful for your entire team. The script prompts you for your Azure subscription information, so have that ready to enter.

   - On a Windows machine, run the script from the PowerShell command prompt:

     ```
     wget "https://raw.githubusercontent.com/Azure/Azure-MachineLearning-
     DataScience/master/Misc/TDSP/CreateFileShare.ps1" -outfile "CreateFileShare.ps1"
     .\CreateFileShare.ps1
     ```

   - On a Linux machine, run the script from the Linux shell:

     ```
     wget "https://raw.githubusercontent.com/Azure/Azure-MachineLearning-
     DataScience/master/Misc/TDSP/CreateFileShare.sh"
     bash CreateFileShare.sh
     ```

2. Log in to your Microsoft Azure account when prompted, and select the subscription you want to use.

3. Select the storage account to use, or create a new one under your selected subscription. You can use lowercase characters, numbers, and hyphens for the Azure file storage name.

4. To facilitate mounting and sharing the storage, press Enter or enter *Y* to save the Azure file storage information into a text file in your current directory. You can check in this text file to your **TeamTemplate** repository, ideally under **Docs\DataDictionaries**, so all projects in your team can access it. You also need the file information to mount your Azure file storage to your Azure DSVM in the next section.

Mount Azure file storage on your local machine or DSVM

1. To mount your Azure file storage to your local machine or DSVM, use the following script.

   - On a Windows machine, run the script from the PowerShell command prompt:

     ```
     wget "https://raw.githubusercontent.com/Azure/Azure-MachineLearning-
     DataScience/master/Misc/TDSP/AttachFileShare.ps1" -outfile "AttachFileShare.ps1"
     .\AttachFileShare.ps1
     ```

   - On a Linux machine, run the script from the Linux shell:

     ```
     wget "https://raw.githubusercontent.com/Azure/Azure-MachineLearning-
     DataScience/master/Misc/TDSP/AttachFileShare.sh"
     bash AttachFileShare.sh
     ```

2. Press Enter or enter *Y* to continue, if you saved an Azure file storage information file in the previous step. Enter the complete path and name of the file you created.

   If you don't have an Azure file storage information file, enter *n*, and follow the instructions to enter your subscription, Azure storage account, and Azure file storage information.

3. Enter the name of a local or TDSP drive to mount the file share on. The screen displays a list of existing drive names. Provide a drive name that doesn't already exist.

4. Confirm that the new drive and storage is successfully mounted on your machine.

Next steps

Here are links to detailed descriptions of the other roles and tasks defined by the Team Data Science Process:

- Group Manager tasks for a data science team
- Project Lead tasks for a data science team
- Project Individual Contributor tasks for a data science team

Project lead tasks in the Team Data Science Process

11/19/2021 • 3 minutes to read • Edit Online

This article describes tasks that a *project lead* completes to set up a repository for their project team in the Team Data Science Process (TDSP). The TDSP is a framework developed by Microsoft that provides a structured sequence of activities to efficiently execute cloud-based, predictive analytics solutions. The TDSP is designed to help improve collaboration and team learning. For an outline of the personnel roles and associated tasks for a data science team standardizing on the TDSP, see Team Data Science Process roles and tasks.

A project lead manages the daily activities of individual data scientists on a specific data science project in the TDSP. The following diagram shows the workflow for project lead tasks:

This tutorial covers Step 1: Create project repository, and Step 2: Seed project repository from your team ProjectTemplate repository.

For Step 3: Create Feature work item for project, and Step 4: Add Stories for project phases, see Agile development of data science projects.

For Step 5: Create and customize storage/analysis assets and share, if necessary, see Create team data and analytics resources.

For Step 6: Set up security control of project repository, see Add team members and configure permissions.

> **NOTE**
>
> This article uses Azure Repos to set up a TDSP project, because that is how to implement TDSP at Microsoft. If your team uses another code hosting platform, the project lead tasks are the same, but the way to complete them may be different.

Prerequisites

This tutorial assumes that your group manager and team lead have set up the following resources and permissions:

- The Azure DevOps **organization** for your data unit
- A team **project** for your data science team
- Team template and utilities **repositories**
- **Permissions** on your organization account for you to create and edit repositories for your project

To clone repositories and modify content on your local machine or Data Science Virtual Machine (DSVM), or set up Azure file storage and mount it to your DSVM, you also need to consider this checklist:

- An Azure subscription.

- Git installed on your machine. If you're using a DSVM, Git is pre-installed. Otherwise, see the Platforms and tools appendix.
- If you want to use a DSVM, the Windows or Linux DSVM created and configured in Azure. For more information and instructions, see the Data Science Virtual Machine Documentation.
- For a Windows DSVM, Git Credential Manager (GCM) installed on your machine. In the *README.md* file, scroll down to the **Download and Install** section and select the **latest installer**. Download the *.exe* installer from the installer page and run it.
- For a Linux DSVM, an SSH public key set up on your DSVM and added in Azure DevOps. For more information and instructions, see the **Create SSH public key** section in the Platforms and tools appendix.

Create a project repository in your team project

To create a project repository in your team's **MyTeam** project:

1. Go to your team's project **Summary** page at *https://<server name>/<organization name>/<team name>*, for example, **https://dev.azure.com/DataScienceUnit/MyTeam**, and select **Repos** from the left navigation.

2. Select the repository name at the top of the page, and then select **New repository** from the dropdown.

3. In the **Create a new repository** dialog, make sure **Git** is selected under **Type**. Enter *DSProject1* under **Repository name**, and then select **Create**.

4. Confirm that you can see the new **DSProject1** repository on your project settings page.

Import the team template into your project repository

To populate your project repository with the contents of your team template repository:

1. From your team's project **Summary** page, select **Repos** in the left navigation.
2. Select the repository name at the top of the page, and select **DSProject1** from the dropdown.
3. On the **DSProject1 is empty** page, select **Import**.

4. In the **Import a Git repository** dialog, select **Git** as the **Source type**, and enter the URL for your **TeamTemplate** repository under **Clone URL**. The URL is *https://<server name>/<organization name>/<team name>/_git/<team template repository name>*. For example: **https://dev.azure.com/DataScienceUnit/MyTeam/_git/TeamTemplate**.

5. Select **Import**. The contents of your team template repository are imported into your project repository.

If you need to customize the contents of your project repository to meet your project's specific needs, you can add, delete, or modify repository files and folders. You can work directly in Azure Repos, or clone the repository to your local machine or DSVM, make changes, and commit and push your updates to the shared project repository. Follow the instructions at Customize the contents of the team repositories.

Next steps

Here are links to detailed descriptions of the other roles and tasks defined by the Team Data Science Process:

- Group Manager tasks for a data science team
- Team Lead tasks for a data science team
- Individual Contributor tasks for a data science team

Tasks for an individual contributor in the Team Data Science Process

11/19/2021 • 3 minutes to read • Edit Online

This topic outlines the tasks that an *individual contributor* completes to set up a project in the Team Data Science Process (TDSP). The objective is to work in a collaborative team environment that standardizes on the TDSP. The TDSP is designed to help improve collaboration and team learning. For an outline of the personnel roles and their associated tasks that are handled by a data science team standardizing on the TDSP, see Team Data Science Process roles and tasks.

The following diagram shows the tasks that project individual contributors (data scientists) complete to set up their team environment. For instructions on how to execute a data science project under the TDSP, see Execution of data science projects.

- **ProjectRepository** is the repository your project team maintains to share project templates and assets.
- **TeamUtilities** is the utilities repository your team maintains specifically for your team.
- **GroupUtilities** is the repository your group maintains to share useful utilities across the entire group.

> **NOTE**
> This article uses Azure Repos and a Data Science Virtual Machine (DSVM) to set up a TDSP environment, because that is how to implement TDSP at Microsoft. If your team uses other code hosting or development platforms, the individual contributor tasks are the same, but the way to complete them may be different.

Prerequisites

This tutorial assumes that the following resources and permissions have been set up by your group manager, team lead, and project lead:

- The Azure DevOps **organization** for your data science unit
- A **project repository** set up by your project lead to share project templates and assets
- **GroupUtilities** and **TeamUtilities** repositories set up by the group manager and team lead, if applicable
- Azure **file storage** set up for shared assets for your team or project, if applicable
- **Permissions** for you to clone from and push back to your project repository

To clone repositories and modify content on your local machine or DSVM, or mount Azure file storage to your DSVM, you need to consider this checklist:

- An Azure subscription.

- Git installed on your machine. If you're using a DSVM, Git is pre-installed. Otherwise, see the Platforms and tools appendix.
- If you want to use a DSVM, the Windows or Linux DSVM created and configured in Azure. For more information and instructions, see the Data Science Virtual Machine Documentation.
- For a Windows DSVM, Git Credential Manager (GCM) installed on your machine. In the *README.md* file, scroll down to the **Download and Install** section and select the **latest installer**. Download the *.exe* installer from the installer page and run it.
- For a Linux DSVM, an SSH public key set up on your DSVM and added in Azure DevOps. For more information and instructions, see the **Create SSH public key** section in the Platforms and tools appendix.
- The Azure file storage information for any Azure file storage you need to mount to your DSVM.

Clone repositories

To work with repositories locally and push your changes up to the shared team and project repositories, you first copy or *clone* the repositories to your local machine.

1. In Azure DevOps, go to your team's project Summary page at *https://<server name>/<organization name>/<team name>*, for example, **https://dev.azure.com/DataScienceUnit/MyTeam**.

2. Select **Repos** in the left navigation, and at the top of the page, select the repository you want to clone.

3. On the repo page, select **Clone** at upper right.

4. In the **Clone repository** dialog, select **HTTPS** for an HTTP connection, or **SSH** for an SSH connection, and copy the clone URL under **Command line** to your clipboard.

5. On your local machine or DSVM, create the following directories:

   - For Windows: **C:\GitRepos**
   - For Linux: **$home/GitRepos**

6. Change to the directory you created.

7. In Git Bash, run the command `git clone <clone URL>` for each repository you want to clone.

   For example, the following command clones the **TeamUtilities** repository to the *MyTeam* directory on your local machine.

HTTPS connection:

```
git clone https://DataScienceUnit@dev.azure.com/DataScienceUnit/MyTeam/_git/TeamUtilities
```

SSH connection:

```
git clone git@ssh.dev.azure.com:v3/DataScienceUnit/MyTeam/TeamUtilities
```

8. Confirm that you can see the folders for the cloned repositories in your local project directory.

```
azureuser@LinuxDSVM1:~/Project_1$
azureuser@LinuxDSVM1:~/Project_1$ dir
GroupUtilities  ProjectRepository  TeamUtilities
```

Mount Azure file storage to your DSVM

If your team or project has shared assets in Azure file storage, mount the file storage to your local machine or DSVM. Follow the instructions at Mount Azure file storage on your local machine or DSVM.

Next steps

Here are links to detailed descriptions of the other roles and tasks defined by the Team Data Science Process:

- Group Manager tasks for a data science team
- Team Lead tasks for a data science team
- Project Lead tasks for a data science team

Team Data Science Process project planning

11/19/2021 • 2 minutes to read • Edit Online

The Team Data Science Process (TDSP) provides a lifecycle to structure the development of your data science projects. This article provides links to Microsoft Project and Excel templates that help you plan and manage these project stages.

The lifecycle outlines the major stages that projects typically execute, often iteratively:

- Business Understanding
- Data Acquisition and Understanding
- Modeling
- Deployment
- Customer Acceptance

For descriptions of each of these stages, see The Team Data Science Process lifecycle.

Microsoft Project template

The Microsoft Project template for the Team Data Science Process is available from here: Microsoft Project template

When you open the plan, click the link to the far left for the TDSP. Change the name and description and then add in any other team resources you need. Estimate the dates required from your experience.

Each task has a note. Open those tasks to see what resources have already been created for you.

Excel template

If don't have access to Microsoft Project, an Excel worksheet with all the same data is also available for download here: Excel template You can pull it in to whatever tool you prefer to use.

Use these templates at your own risk. The usual disclaimers apply.

Repository template

Use this project template repository to support efficient project execution and collaboration. This repository gives you a standardized directory structure and document templates you can use for your own TDSP project.

Next steps

Agile development of data science projects This document describes a data science project in a systematic, version controlled, and collaborative way by using the Team Data Science Process.

Walkthroughs that demonstrate all the steps in the process for **specific scenarios** are also provided. They are listed and linked with thumbnail descriptions in the Example walkthroughs article. They illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

Agile development of data science projects

11/19/2021 • 7 minutes to read • Edit Online

This document describes how developers can execute a data science project in a systematic, version controlled, and collaborative way within a project team by using the Team Data Science Process (TDSP). The TDSP is a framework developed by Microsoft that provides a structured sequence of activities to efficiently execute cloud-based, predictive analytics solutions. For an outline of the roles and tasks that are handled by a data science team standardizing on the TDSP, see Team Data Science Process roles and tasks.

This article includes instructions on how to:

- Do *sprint planning* for work items involved in a project.
- Add *work items* to sprints.
- Create and use an *agile-derived work item template* that specifically aligns with TDSP lifecycle stages.

The following instructions outline the steps needed to set up a TDSP team environment using Azure Boards and Azure Repos in Azure DevOps. The instructions use Azure DevOps because that is how to implement TDSP at Microsoft. If your group uses a different code hosting platform, the team lead tasks generally don't change, but the way to complete the tasks is different. For example, linking a work item with a Git branch might not be the same with GitHub as it is with Azure Repos.

The following figure illustrates a typical sprint planning, coding, and source-control workflow for a data science project:

Work item types

In the TDSP sprint planning framework, there are four frequently used *work item* types: *Features*, *User Stories*, *Tasks*, and *Bugs*. The backlog for all work items is at the project level, not the Git repository level.

Here are the definitions for the work item types:

- **Feature**: A Feature corresponds to a project engagement. Different engagements with a client are different Features, and it's best to consider different phases of a project as different Features. If you choose a schema such as *<ClientName>-<EngagementName>* to name your Features, you can easily recognize the context of the project and engagement from the names themselves.
- **User Story**: User Stories are work items needed to complete a Feature end-to-end. Examples of User Stories include:

    - Get data
    - Explore data
    - Generate features
    - Build models
    - Operationalize models
    - Retrain models
- **Task**: Tasks are assignable work items that need to be done to complete a specific User Story. For example, Tasks in the User Story *Get data* could be:
    - Get SQL Server credentials
    - Upload data to Azure Synapse Analytics
- **Bug**: Bugs are issues in existing code or documents that must be fixed to complete a Task. If Bugs are caused by missing work items, they can escalate to be User Stories or Tasks.

Data scientists may feel more comfortable using an agile template that replaces Features, User Stories, and Tasks with TDSP lifecycle stages and substages. To create an agile-derived template that specifically aligns with the TDSP lifecycle stages, see Use an agile TDSP work template.

> **NOTE**
> TDSP borrows the concepts of Features, User Stories, Tasks, and Bugs from software code management (SCM). The TDSP concepts might differ slightly from their conventional SCM definitions.

Plan sprints

Many data scientists are engaged with multiple projects, which can take months to complete and proceed at different paces. Sprint planning is useful for project prioritization, and resource planning and allocation. In Azure Boards, you can easily create, manage, and track work items for your projects, and conduct sprint planning to ensure projects are moving forward as expected.

For more information about sprint planning, see Scrum sprints.

For more information about sprint planning in Azure Boards, see Assign backlog items to a sprint.

Add a Feature to the backlog

After your project and project code repository are created, you can add a Feature to the backlog to represent the work for your project.

1. From your project page, select **Boards** > **Backlogs** in the left navigation.
2. On the **Backlog** tab, if the work item type in the top bar is **Stories**, drop down and select **Features**. Then select **New Work Item**.

3. Enter a title for the Feature, usually your project name, and then select **Add to top**.

4. From the **Backlog** list, select and open the new Feature. Fill in the description, assign a team member, and set planning parameters.

   You can also link the Feature to the project's Azure Repos code repository by selecting **Add link** under the **Development** section.

   After you edit the Feature, select **Save & Close**.

Add a User Story to the Feature

Under the Feature, you can add User Stories to describe major steps needed to complete the project.

To add a new User Story to a Feature:

1. On the **Backlog** tab, select the **+** to the left of the Feature.

2. Give the User Story a title, and edit details such as assignment, status, description, comments, planning, and priority.

   You can also link the User Story to a branch of the project's Azure Repos code repository by selecting **Add link** under the **Development** section. Select the repository and branch you want to link the work item to, and then select **OK**.

3. When you're finished editing the User Story, select **Save & Close**.

Add a Task to a User Story

Tasks are specific detailed steps that are needed to complete each User Story. After all Tasks of a User Story are completed, the User Story should be completed too.

To add a Task to a User Story, select the **+** next to the User Story item, and select **Task**. Fill in the title and other information in the Task.

After you create Features, User Stories, and Tasks, you can view them in the **Backlogs** or **Boards** views to track their status.

Use an agile TDSP work template

Data scientists may feel more comfortable using an agile template that replaces Features, User Stories, and Tasks with TDSP lifecycle stages and substages. In Azure Boards, you can create an agile-derived template that uses TDSP lifecycle stages to create and track work items. The following steps walk through setting up a data science-specific agile process template and creating data science work items based on the template.

Set up an Agile Data Science Process template

1. From your Azure DevOps organization main page, select **Organization settings** from the left navigation.

2. In the **Organization Settings** left navigation, under **Boards**, select **Process**.

3. In the **All processes** pane, select the **...** next to **Agile**, and then select **Create inherited process**.

4. In the **Create inherited process from Agile** dialog, enter the name *AgileDataScienceProcess*, and select **Create process**.

5. In **All processes**, select the new **AgileDataScienceProcess**.

6. On the **Work item types** tab, disable **Epic**, **Feature**, **User Story**, and **Task** by selecting the **...** next to each item and then selecting **Disable**.

7. In **All processes**, select the **Backlog levels** tab. Under **Portfolios backlogs**, select the **...** next to **Epic (disabled)**, and then select **Edit/Rename**.

8. In the **Edit backlog level** dialog box:

   a. Under **Name**, replace **Epic** with *TDSP Projects*.
   b. Under **Work item types on this backlog level**, select **New work item type**, enter *TDSP Project*, and select **Add**.
   c. Under **Default work item type**, drop down and select **TDSP Project**.
   d. Select **Save**.

9. Follow the same steps to rename **Features** to *TDSP Stages*, and add the following new work item types:

- *Business Understanding*
- *Data Acquisition*
- *Modeling*
- *Deployment*

10. Under **Requirement backlog**, rename **Stories** to *TDSP Substages*, add the new work item type *TDSP Substage*, and set the default work item type to **TDSP Substage**.

11. Under **Iteration backlog**, add a new work item type *TDSP Task*, and set it to be the default work item type.

After you complete the steps, the backlog levels should look like this:

Create Agile Data Science Process work items

You can use the data science process template to create TDSP projects and track work items that correspond to TDSP lifecycle stages.

1. From your Azure DevOps organization main page, select **New project**.

2. In the **Create new project** dialog, give your project a name, and then select **Advanced**.

3. Under **Work item process**, drop down and select **AgileDataScienceProcess**, and then select **Create**.

4. In the newly created project, select **Boards** > **Backlogs** in the left navigation.

5. To make TDSP Projects visible, select the **Configure team settings** icon. In the **Settings** screen, select the **TDSP Projects** check box, and then select **Save and close**.

6. To create a data science-specific TDSP Project, select **TDSP Projects** in the top bar, and then select **New work item**.

7. In the popup, give the TDSP Project work item a name, and select **Add to top**.

8. To add a work item under the TDSP Project, select the + next to the project, and then select the type of work item to create.

9. Fill in the details in the new work item, and select **Save & Close**.

10. Continue to select the + symbols next to work items to add new TDSP Stages, Substages, and Tasks.

Here is an example of how the data science project work items should appear in **Backlogs** view:

Next steps

Collaborative coding with Git describes how to do collaborative code development for data science projects using Git as the shared code development framework, and how to link these coding activities to the work planned with the agile process.

Example walkthroughs lists walkthroughs of specific scenarios, with links and thumbnail descriptions. The linked scenarios illustrate how to combine cloud and on-premises tools and services into workflows or pipelines to create intelligent applications.

Additional resources on agile processes:

- Agile process
- Agile process work item types and workflow

Collaborative coding with Git

11/19/2021 • 4 minutes to read • Edit Online

This article describes how to use Git as the collaborative code development framework for data science projects. The article covers how to link code in Azure Repos to agile development work items in Azure Boards, how to do code reviews, and how to create and merge pull requests for changes.

Link a work item to an Azure Repos branch

Azure DevOps provides a convenient way to connect an Azure Boards User Story or Task work item with an Azure Repos Git repository branch. You can link your User Story or Task directly to the code associated with it.

To connect a work item to a new branch, select the **Actions** ellipsis (...) next to the work item, and on the context menu, scroll to and select **New branch**.

In the **Create a branch** dialog, provide the new branch name and the base Azure Repos Git repository and branch. The base repository must be in the same Azure DevOps project as the work item. The base branch can be any existing branch. Select **Create branch**.

You can also create a new branch using the following Git bash command in Windows or Linux:

```
git checkout -b <new branch name> <base branch name>
```

If you don't specify a <base branch name>, the new branch is based on `main`.

To switch to your working branch, run the following command:

```
git checkout <working branch name>
```

After you switch to the working branch, you can start developing code or documentation artifacts to complete the work item. Running `git checkout main` switches you back to the `main` branch.

It's a good practice to create a Git branch for each User Story work item. Then, for each Task work item, you can create a branch based on the User Story branch. Organize the branches in a hierarchy that corresponds to the User Story-Task relationship when you have multiple people working on different User Stories for the same project, or on different Tasks for the same User Story. You can minimize conflicts by having each team member work on a different branch, or on different code or other artifacts when sharing a branch.

The following diagram shows the recommended branching strategy for TDSP. You might not need as many branches as shown here, especially when only one or two people work on a project, or only one person works on all Tasks of a User Story. But separating the development branch from the primary branch is always a good practice, and can help prevent the release branch from being interrupted by development activities. For a complete description of the Git branch model, see A Successful Git Branching Model.

You can also link a work item to an existing branch. On the **Detail** page of a work item, select **Add link**. Then select an existing branch to link the work item to, and select **OK**.

Work on the branch and commit changes

After you make a change for your work item, such as adding an R script file to your local machine's `script` branch, you can commit the change from your local branch to the upstream working branch by using the following Git bash commands:

```
git status
git add .
git commit -m "added an R script file"
git push origin script
```

Create a pull request

After one or more commits and pushes, when you're ready to merge your current working branch into its base branch, you can create and submit a *pull request* in Azure Repos.

From the main page of your Azure DevOps project, point to **Repos** > **Pull requests** in the left navigation. Then select either of the **New pull request** buttons, or the **Create a pull request** link.

On the **New Pull Request** screen, if necessary, navigate to the Git repository and branch you want to merge your changes into. Add or change any other information you want. Under **Reviewers**, add the names of the reviewers, and then select **Create**.

Review and merge

Once you create the pull request, your reviewers get an email notification to review the pull request. The reviewers test whether the changes work, and check the changes with the requester if possible. The reviewers can make comments, request changes, and approve or reject the pull request based on their assessment.

After the reviewers approve the changes, you or someone else with merge permissions can merge the working branch to its base branch. Select **Complete**, and then select **Complete merge** in the **Complete pull request** dialog. You can choose to delete the working branch after it has merged.

Confirm that the request is marked as **COMPLETED**.

When you go back to **Repos** in the left navigation, you can see that you've been switched to the main branch since the `script` branch was deleted.

You can also use the following Git bash commands to merge the `script` working branch to its base branch and delete the working branch after merging:

```
git checkout main
git merge script
git branch -d script
```

Next steps

Execute data science tasks shows how to use utilities to complete several common data science tasks, such as interactive data exploration, data analysis, reporting, and model creation.

Example walkthroughs lists walkthroughs of specific scenarios, with links and thumbnail descriptions. The linked scenarios illustrate how to combine cloud and on-premises tools and services into workflows or pipelines to create intelligent applications.

Execute data science tasks: exploration, modeling, and deployment

11/19/2021 • 2 minutes to read • Edit Online

Typical data science tasks include data exploration, modeling, and deployment. This article outlines the tasks to complete several common data science tasks such as interactive data exploration, data analysis, reporting, and model creation. Options for deploying a model into a production environment may include:

- Azure Machine Learning
- SQL-Server with ML services
- Microsoft Machine Learning Server

1. Exploration

A data scientist can perform exploration and reporting in a variety of ways: by using libraries and packages available for Python (matplotlib for example) or with R (ggplot or lattice for example). Data scientists can customize such code to fit the needs of data exploration for specific scenarios. The needs for dealing with structured data are different that for unstructured data such as text or images.

Products such as Azure Machine Learning also provide advanced data preparation for data wrangling and exploration, including feature creation. The user should decide on the tools, libraries, and packages that best suite their needs.

The deliverable at the end of this phase is a data exploration report. The report should provide a fairly comprehensive view of the data to be used for modeling and an assessment of whether the data is suitable to proceed to the modeling step.

2. Modeling

There are numerous toolkits and packages for training models in a variety of languages. Data scientists should feel free to use which ever ones they are comfortable with, as long as performance considerations regarding accuracy and latency are satisfied for the relevant business use cases and production scenarios.

Model management

After multiple models have been built, you usually need to have a system for registering and managing the models. Typically you need a combination of scripts or APIs and a backend database or versioning system. A few options that you can consider for these management tasks are:

1. Azure Machine Learning - model management service
2. ModelDB from MIT
3. SQL-server as a model management system
4. Microsoft Machine Learning Server

3. Deployment

Production deployment enables a model to play an active role in a business. Predictions from a deployed model

can be used for business decisions.

Production platforms

There are various approaches and platforms to put models into production. Here are a few options:

- Model deployment in Azure Machine Learning
- Deployment of a model in SQL-server
- Microsoft Machine Learning Server

> **NOTE**
>
> Prior to deployment, one has to insure the latency of model scoring is low enough to use in production.

Further examples are available in walkthroughs that demonstrate all the steps in the process for **specific scenarios**. They are listed and linked with thumbnail descriptions in the Example walkthroughs article. They illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

> **NOTE**
>
> For deployment using Machine Learning Studio (classic), see Deploy a Machine Learning Studio (classic) web service.

A/B testing

When multiple models are in production, it can be useful to perform A/B testing to compare performance of the models.

Next steps

Track progress of data science projects shows how a data scientist can track the progress of a data science project.

Model operation and CI/CD shows how CI/CD can be performed with developed models.

Test data science code with Azure DevOps

11/19/2021 • 4 minutes to read • Edit Online

This article gives preliminary guidelines for testing code in a data science workflow, using Azure DevOps. Such testing gives data scientists a systematic and efficient way to check the quality and expected outcome of their code. We use a Team Data Science Process (TDSP) project that uses the UCI Adult Income dataset that we published earlier to show how code testing can be done.

Introduction on code testing

"Unit testing" is a longstanding practice for software development. But for data science, it's often not clear what "unit testing" means and how you should test code for different stages of a data science lifecycle, such as:

- Data preparation
- Data quality examination
- Modeling
- Model deployment

This article replaces the term "unit testing" with "code testing." It refers to testing as the functions that help to assess if code for a certain step of a data science lifecycle is producing results "as expected." The person who's writing the test defines what's "as expected," depending on the outcome of the function--for example, data quality check or modeling.

This article provides references as useful resources.

Azure DevOps for the testing framework

This article describes how to perform and automate testing by using Azure DevOps. You might decide to use alternative tools. We also show how to set up an automatic build by using Azure DevOps and build agents. For build agents, we use Azure Data Science Virtual Machines (DSVMs).

Flow of code testing

The overall workflow of testing code in a data science project looks like this:

Detailed steps

Use the following steps to set up and run code testing and an automated build by using a build agent and Azure DevOps:

1. Create a project in the Visual Studio desktop application:

After you create your project, you'll find it in Solution Explorer in the right pane:

2. Feed your project code into the Azure DevOps project code repository:

3. Suppose you've done some data preparation work, such as data ingestion, feature engineering, and creating label columns. You want to make sure your code is generating the results that you expect. Here's some code that you can use to test whether the data-processing code is working properly:

   - Check that column names are right:

```
def checkColumnNames(csv_file, required_columns):
    df = pd.read_csv(csv_file)
    if set(df.columns) == set(required_columns):
        print("Columns match!")
    else:
        print("Columns do not matched!")
    return set(df.columns) == set(required_columns)
```

   Check that response levels are right:

```
def checkResponseLevels(csv_file,response_column):
    df = pd.read_csv(csv_file)
    levels=list(df[response_column].unique())
    if levels == [0,1]:
        print("Levels match!")
    else:
        print("Levels do not match!")
    return levels == [0,1]
```

   - Check that response percentage is reasonable:

```python
def checkResponsePercent(csv_file,response_column):
    df = pd.read_csv(csv_file)
    ratio1 = df.loc[df[response_column] == 1].shape[0]/df.shape[0]
    if ratio1 > 0.5:
        print("Response levels(0/1) are messed up!")
    else:
        print("Response 0/1 are OK, and OK, Happy Friday!")
    return ratio1 < 0.5
```

- Check the missing rate of each column in the data:

```python
def checkMissingRate2(csv_file, threshold):
    df = pd.read_csv(csv_file)
    for x in df.columns:
        miss_rate = df[x].isnull().sum()/df.shape[0]
        if miss_rate > threshold:
            print("{} has more than {} missing values!".format(x,threshold))
            return False
    print("No column has missing values more than {}!".format(threshold))
    return True
```

4. After you've done the data processing and feature engineering work, and you've trained a good model, make sure that the model you trained can score new datasets correctly. You can use the following two tests to check the prediction levels and distribution of label values:

- Check prediction levels:

```python
def checkPredictionLevels(csv_file, model_file):
    df = pd.read_csv(csv_file)
    X_to_score = df[['education_num','age','hours_per_week']].values
    loaded_model = pickle.load(open(model_file, 'rb'))
    y_hat = loaded_model.predict(X_to_score)
    if list(set(y_hat)) == [0,1]:
        print("Prediction Levels match!")
    else:
        print("Prediction Levels do not match!")
    return list(set(y_hat)) == [0,1]
```

- Check the distribution of prediction values:

```python
def checkPredictionPercent(csv_file,model_file):
    df = pd.read_csv(csv_file)
    X_to_score = df[['education_num','age','hours_per_week']].values
    loaded_model = pickle.load(open(model_file, 'rb'))
    y_hat = loaded_model.predict(X_to_score)
    ratio1 = sum(y_hat == 1)/len(y_hat)
    if ratio1 > 0.5:
        print("Response levels(0/1) are messed up!")
    else:
        print("Response 0/1 percent is OK, Happy prediction!")
    return ratio1 < 0.5
```

5. Put all test functions together into a Python script called **test_funcs.py**:

test_funcs.py

```python
#class check_data(object):
#    """description of class"""
import pandas as pd
import pickle
import numpy as np


def checkColumnNames(csv_file, required_columns):
    df = pd.read_csv(csv_file)
    if set(df.columns) == set(required_columns):
        print("Columns match!")
    else:
        print("Columns do not matched!")
    return set(df.columns) == set(required_columns)

def checkResponseLevels(csv_file,response_column):
    df = pd.read_csv(csv_file)
    levels=list(df[response_column].unique())
    if levels == [0,1]:
        print("Levels match!")
    else:
        print("Levels do not match!")
    return levels == [0,1]

def checkResponsePercent(csv_file,response_column):
    df = pd.read_csv(csv_file)
    ratio1 = df.loc[df[response_column] == 1].shape[0]/df.shape[0]
    if ratio1 > 0.5:
        print("Response levels(0/1) are messed up!")
    else:
        print("Response 0/1 are OK, and OK, Happy Friday!")
    return ratio1 < 0.5
```

6. After the test codes are prepared, you can set up the testing environment in Visual Studio.

   Create a Python file called **test1.py**. In this file, create a class that includes all the tests you want to do. The following example shows six tests prepared:

```python
class Test_A(unittest.TestCase):

    def checkColumnNames(self):
        assert checkColumnNames(file_name, required_columns) == True

    def checkResponseLevels(self):
        assert checkResponseLevels(file_name, response_column) == True

    def checkResponsePercent(self):
        assert checkResponsePercent(file_name, response_column) == True

    def checkMissingRate(self):
        assert checkMissingRate2(file_name, threshold) == True

    def checkPredictionLevels(self):
        assert checkPredictionLevels(score_file_name, model_file_name) == True

    def checkPredictionPercent(self):
        assert checkPredictionPercent(score_file_name, model_file_name) == True
```

1. Those tests can be automatically discovered if you put **codetest.testCase** after your class name. Open Test Explorer in the right pane, and select **Run All**. All the tests will run sequentially and will tell you if the test is successful or not.

2. Check in your code to the project repository by using Git commands. Your most recent work will be reflected shortly in Azure DevOps.

```
PS C:\Unit_Test_UCI_Income> git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   .vs/unit_test_1/v15/.suo
        modified:   __pycache__/test_funcs.cpython-35.pyc
        modified:   test1.py
        modified:   test1.pyc
        modified:   test_funcs.py
        modified:   unit_test.pyproj

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        __pycache__/test1.cpython-35.pyc
        __pycache__/test2.cpython-35.pyc
        __pycache__/test_model.cpython-35.pyc

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Unit_Test_UCI_Income> git add .
PS C:\Unit_Test_UCI_Income> git commit -m"revised code wei"
[master 60841e1] revised code wei
 9 files changed, 7 insertions(+), 19 deletions(-)
 rewrite .vs/unit_test_1/v15/.suo (63%)
 create mode 100644 __pycache__/test1.cpython-35.pyc
 create mode 100644 __pycache__/test2.cpython-35.pyc
 create mode 100644 __pycache__/test_model.cpython-35.pyc
 rewrite test1.pyc (64%)
PS C:\Unit_Test_UCI_Income> git push
Counting objects: 15, done.
Delta compression using up to 24 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (15/15), 6.16 KiB | 2.05 MiB/s, done.
Total 15 (delta 6), reused 0 (delta 0)
remote: Analyzing objects... (15/15) (26 ms)
remote: Storing packfile... done (213 ms)
remote: Storing index... done (32 ms)
To https://dg-ads.visualstudio.com/_git/Unit_Test_UCI_Income
   7785cf0..60841e1  master -> master
PS C:\Unit_Test_UCI_Income>
```

3. Set up automatic build and test in Azure DevOps:

   a. In the project repository, select **Build and Release**, and then select **+New** to create a new build process.

   b. Follow the prompts to select your source code location, project name, repository, and branch information.

   c. Select a template. Because there's no Python project template, start by selecting **Empty process**.

d. Name the build and select the agent. You can choose the default here if you want to use a DSVM to complete the build process. For more information about setting agents, see Build and release agents.

e. Select **+** in the left pane, to add a task for this build phase. Because we're going to run the Python script **test1.py** to complete all the checks, this task is using a PowerShell command to run Python code.

f. In the PowerShell details, fill in the required information, such as the name and version of PowerShell. Choose **Inline Script** as the type.

In the box under **Inline Script**, you can type **python test1.py**. Make sure the environment variable is set up correctly for Python. If you need a different version or kernel of Python, you can explicitly specify the path as shown in the figure:

g. Select **Save & queue** to complete the build pipeline process.

Now every time a new commit is pushed to the code repository, the build process will start automatically. You can define any branch. The process runs the **test1.py** file in the agent machine to make sure that everything defined in the code runs correctly.

If alerts are set up correctly, you'll be notified in email when the build is finished. You can also check the build status in Azure DevOps. If it fails, you can check the details of the build and find out which piece is broken.

Next steps

- See the UCI income prediction repository for concrete examples of unit tests for data science scenarios.
- Follow the preceding outline and examples from the UCI income prediction scenario in your own data science projects.

References

- Team Data Science Process
- Visual Studio Testing Tools
- Azure DevOps Testing Resources
- Data Science Virtual Machines

Track the progress of data science projects

11/19/2021 • 2 minutes to read • Edit Online

Data science group managers, team leads, and project leads can track the progress of their projects. Managers want to know what work has been done, who did the work, and what work remains. Managing expectations is an important element of success.

Azure DevOps dashboards

If you're using Azure DevOps, you can build dashboards to track the activities and work items associated with a given Agile project. For more information about dashboards, see Dashboards, reports, and widgets.

For instructions on how to create and customize dashboards and widgets in Azure DevOps, see the following quickstarts:

- Add and manage dashboards
- Add widgets to a dashboard

Example dashboard

Here is a simple example dashboard that tracks the sprint activities of an Agile data science project, including the number of commits to associated repositories.

- The **countdown** tile shows the number of days that remain in the current sprint.
- The two **code tiles** show the number of commits in the two project repositories for the past seven days.
- **Work items for TDSP Customer Project** shows the results of a query for all work items and their status.
- A **cumulative flow diagram** (CFD) shows the number of Closed and Active work items.
- The **burndown chart** shows work still to complete against remaining time in the sprint.
- The **burnup chart** shows completed work compared to total amount of work in the sprint.

Next steps

Walkthroughs executing the Team Data Science Process lists walkthroughs that demonstrate all the process steps. The linked scenarios illustrate how to manage the cloud and on-premise resources into intelligent applications.

Create CI/CD pipelines for AI apps using Azure Pipelines, Docker, and Kubernetes

11/19/2021 • 2 minutes to read • Edit Online

An Artificial Intelligence (AI) application is application code embedded with a pretrained machine learning (ML) model. There are always two streams of work for an AI application: Data scientists build the ML model, and app developers build the app and expose it to end users to consume. This article describes how to implement a continuous integration and continuous delivery (CI/CD) pipeline for an AI application that embeds the ML model into the app source code. The sample code and tutorial use a Python Flask web application, and fetch a pretrained model from a private Azure blob storage account. You could also use an AWS S3 storage account.

> **NOTE**
>
> The following process is one of several ways to do CI/CD. There are alternatives to this tooling and the prerequisites.

Source code, tutorial, and prerequisites

You can download source code and a detailed tutorial from GitHub. Follow the tutorial steps to implement a CI/CD pipeline for your own application.

To use the downloaded source code and tutorial, you need the following prerequisites:

- The source code repository forked to your GitHub account
- An Azure DevOps Organization
- Azure CLI
- An Azure Container Service for Kubernetes (AKS) cluster
- Kubectl to run commands and fetch configuration from the AKS cluster
- An Azure Container Registry (ACR) account

CI/CD pipeline summary

Each new Git commit kicks off the Build pipeline. The build securely pulls the latest ML model from a blob storage account, and packages it with the app code in a single container. This decoupling of the application development and data science workstreams ensures that the production app is always running the latest code with the latest ML model. If the app passes testing, the pipeline securely stores the build image in a Docker container in ACR. The release pipeline then deploys the container using AKS.

CI/CD pipeline steps

The following diagram and steps describe the CI/CD pipeline architecture:

1. Developers work on the application code in the IDE of their choice.
2. The developers commit the code to Azure Repos, GitHub, or other Git source control provider.
3. Separately, data scientists work on developing their ML model.
4. The data scientists publish the finished model to a model repository, in this case a blob storage account.
5. Azure Pipelines kicks off a build based on the Git commit.
6. The Build pipeline pulls the latest ML model from blob storage and creates a container.
7. The pipeline pushes the build image to the private image repository in ACR.
8. The Release pipeline kicks off based on the successful build.
9. The pipeline pulls the latest image from ACR and deploys it across the Kubernetes cluster on AKS.
10. User requests for the app go through the DNS server.
11. The DNS server passes the requests to a load balancer, and sends responses back to the users.

See also

- Team Data Science Process (TDSP)
- Azure Machine Learning (AML)
- Azure DevOps
- Azure Kubernetes Services (AKS)

Walkthroughs executing the Team Data Science Process

11/19/2021 • 2 minutes to read • Edit Online

These **comprehensive walkthroughs** demonstrate the steps in the Team Data Science Process for specific scenarios. They illustrate how to combine cloud, on-premises tools, and services into a workflow for an **intelligent application**. The walkthroughs are grouped by **platform** that they use.

Walkthrough descriptions

Here are brief descriptions of what these walkthrough examples provide on their respective platforms:

- HDInsight Spark walkthroughs using PySpark and Scala These walkthroughs use PySpark and Scala on an Azure Spark cluster to do predictive analytics.
- HDInsight Hadoop walkthroughs using Hive These walkthroughs use Hive with an HDInsight Hadoop cluster to do predictive analytics.
- Azure Data Lake walkthroughs using U-SQL These walkthroughs use U-SQL with Azure Data Lake to do predictive analytics.
- SQL Server These walkthroughs use SQL Server, SQL Server R Services, and SQL Server Python Services to do predictive analytics.
- Azure Synapse Analytics These walkthroughs use Azure Synapse Analytics to do predictive analytics.

Next steps

For a discussion of the key components that comprise the Team Data Science Process, see Team Data Science Process overview.

For a discussion of the Team Data Science Process lifecycle, see Team Data Science Process lifecycle. This lifecycle outlines the steps, from start to finish, that projects usually follow when they are executed.

For an overview, see Data Science Process.

HDInsight Spark data science walkthroughs using PySpark and Scala on Azure

11/19/2021 • 2 minutes to read • Edit Online

These walkthroughs use PySpark and Scala on an Azure Spark cluster to do predictive analytics. They follow the steps outlined in the Team Data Science Process. For an overview of the Team Data Science Process, see Data Science Process. For an overview of Spark on HDInsight, see Introduction to Spark on HDInsight.

Additional data science walkthroughs that execute the Team Data Science Process are grouped by the **platform** that they use. See Walkthroughs executing the Team Data Science Process for an itemization of these examples.

Predict taxi tips using PySpark on Azure Spark

Using New York taxi data, the Use Spark on Azure HDInsight walkthrough predicts whether a tip is paid and the range of expected amounts. This example uses the Team Data Science Process in a scenario using an Azure HDInsight Spark cluster to store, explore, and feature engineer data from the publicly available NYC taxi trip and fare dataset. This overview topic uses an HDInsight Spark cluster and Jupyter PySpark notebooks. These notebooks show you how to explore your data and then how to create and consume models. The advanced data exploration and modeling notebook shows how to include cross-validation, hyper-parameter sweeping, and model evaluation.

Data Exploration and modeling with Spark

Explore the dataset and create, score, and evaluate the machine learning models by working through the Create binary classification and regression models for data with the Spark MLlib toolkit topic.

Model consumption

To learn how to score the classification and regression models created in this topic, see Score and evaluate Spark-built machine learning models.

Cross-validation and hyperparameter sweeping

See Advanced data exploration and modeling with Spark on how models can be trained using cross-validation and hyper-parameter sweeping.

Predict taxi tips using Scala on Azure Spark

The Use Scala with Spark on Azure walkthrough predicts whether a tip is paid and the range of amounts expected to be paid. It shows how to use Scala for supervised machine learning tasks with the Spark machine learning library (MLlib) and SparkML packages on an Azure HDInsight Spark cluster. It walks you through the tasks that constitute the Data Science Process: data ingestion and exploration, visualization, feature engineering, modeling, and model consumption. The models built include logistic and linear regression, random forests, and gradient boosted trees.

Next steps

For an overview of the Team Data Science Process, see Team Data Science Process overview.

For a discussion of the Team Data Science Process lifecycle, see Team Data Science Process lifecycle. This lifecycle outlines the steps, from start to finish, that projects usually follow when they are executed.

Data exploration and modeling with Spark

11/19/2021 • 27 minutes to read • Edit Online

Learn how to use HDInsight Spark to train machine learning models for taxi fare prediction using Spark MLlib.

This sample showcases the various steps in the Team Data Science Process. A subset of the NYC taxi trip and fare 2013 dataset is used to load, explore and prepare data. Then, using Spark MLlib, binary classification and regression models are trained to predict whether a tip will be paid for the trip and estimate the tip amount.

Prerequisites

You need an Azure account and a Spark 1.6 (or Spark 2.0) HDInsight cluster to complete this walkthrough. See the Overview of Data Science using Spark on Azure HDInsight for instructions on how to satisfy these requirements. That topic also contains a description of the NYC 2013 Taxi data used here and instructions on how to execute code from a Jupyter notebook on the Spark cluster.

Spark clusters and notebooks

Setup steps and code are provided in this walkthrough for using an HDInsight Spark 1.6. But Jupyter notebooks are provided for both HDInsight Spark 1.6 and Spark 2.0 clusters. A description of the notebooks and links to them are provided in the Readme.md for the GitHub repository containing them. Moreover, the code here and in the linked notebooks is generic and should work on any Spark cluster. If you are not using HDInsight Spark, the cluster setup and management steps may be slightly different from what is shown here. For convenience, here are the links to the Jupyter notebooks for Spark 1.6 (to be run in the pySpark kernel of the Jupyter Notebook server) and Spark 2.0 (to be run in the pySpark3 kernel of the Jupyter Notebook server):

- Spark 1.6 notebooks: Provide information on how to perform data exploration, modeling, and scoring with several different algorithms.
- Spark 2.0 notebooks: Provide information on how to perform regression and classification tasks. Datasets may vary, but the steps and concepts are applicable to various datasets.

> **WARNING**
> Billing for HDInsight clusters is prorated per minute, whether you use them or not. Be sure to delete your cluster after you finish using it. See how to delete an HDInsight cluster.

> **NOTE**
> The descriptions below are related to using Spark 1.6. For Spark 2.0 versions, please use the notebooks described and linked above.

Setup

Spark is able to read and write to Azure Storage Blob (also known as WASB). So any of your existing data stored there can be processed using Spark and the results stored again in WASB.

To save models or files in WASB, the path needs to be specified properly. The default container attached to the Spark cluster can be referenced using a path beginning with: "wasb:///". Other locations are referenced by "wasb://".

Set directory paths for storage locations in WASB

The following code sample specifies the location of the data to be read and the path for the model storage directory to which the model output is saved:

```
# SET PATHS TO FILE LOCATIONS: DATA AND MODEL STORAGE

# LOCATION OF TRAINING DATA
taxi_train_file_loc =
"wasb://mllibwalkthroughs@cdspsparksamples.blob.core.windows.net/Data/NYCTaxi/JoinedTaxiTripFare.Point1Pct.T
rain.tsv";

# SET THE MODEL STORAGE DIRECTORY PATH
# NOTE THAT THE FINAL BACKSLASH IN THE PATH IS NEEDED.
modelDir = "wasb:///user/remoteuser/NYCTaxi/Models/"
```

Import libraries

Set up also requires importing necessary libraries. Set spark context and import necessary libraries with the following code:

```
# IMPORT LIBRARIES
import pyspark
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.sql import SQLContext
import matplotlib
import matplotlib.pyplot as plt
from pyspark.sql import Row
from pyspark.sql.functions import UserDefinedFunction
from pyspark.sql.types import *
import atexit
from numpy import array
import numpy as np
import datetime
```

Preset Spark context and PySpark magics

The PySpark kernels that are provided with Jupyter notebooks have a preset context. So you do not need to set the Spark or Hive contexts explicitly before you start working with the application you are developing. These contexts are available for you by default. These contexts are:

- sc - for Spark
- sqlContext - for Hive

The PySpark kernel provides some predefined "magics", which are special commands that you can call with %%. There are two such commands that are used in these code samples.

- **%%local** Specifies that the code in subsequent lines is to be executed locally. Code must be valid Python code.
- **%%sql -o <variable name>** Executes a Hive query against the sqlContext. If the -o parameter is passed, the result of the query is persisted in the %%local Python context as a Pandas DataFrame.

For more information on Jupyter notebook kernels and the predefined "magics", see Kernels available for Jupyter notebooks with HDInsight Spark Linux clusters on HDInsight.

Load the data

The first step in the data science process is to ingest the data to be analyzed from sources where is resides into your data exploration and modeling environment. The environment is Spark in this walkthrough. This section contains the code to complete a series of tasks:

- ingest the data sample to be modeled
- read in the input dataset (stored as a .tsv file)
- format and clean the data
- create and cache objects (RDDs or data-frames) in memory
- register it as a temp-table in SQL-context.

Here is the code for data ingestion.

```
# INGEST DATA

# RECORD START TIME
timestart = datetime.datetime.now()

# IMPORT FILE FROM PUBLIC BLOB
taxi_train_file = sc.textFile(taxi_train_file_loc)

# GET SCHEMA OF THE FILE FROM HEADER
schema_string = taxi_train_file.first()
fields = [StructField(field_name, StringType(), True) for field_name in schema_string.split('\t')]
fields[7].dataType = IntegerType() #Pickup hour
fields[8].dataType = IntegerType() # Pickup week
fields[9].dataType = IntegerType() # Weekday
fields[10].dataType = IntegerType() # Passenger count
fields[11].dataType = FloatType() # Trip time in secs
fields[12].dataType = FloatType() # Trip distance
fields[19].dataType = FloatType() # Fare amount
fields[20].dataType = FloatType() # Surcharge
fields[21].dataType = FloatType() # Mta_tax
fields[22].dataType = FloatType() # Tip amount
fields[23].dataType = FloatType() # Tolls amount
fields[24].dataType = FloatType() # Total amount
fields[25].dataType = IntegerType() # Tipped or not
fields[26].dataType = IntegerType() # Tip class
taxi_schema = StructType(fields)

# PARSE FIELDS AND CONVERT DATA TYPE FOR SOME FIELDS
taxi_header = taxi_train_file.filter(lambda l: "medallion" in l)
taxi_temp = taxi_train_file.subtract(taxi_header).map(lambda k: k.split("\t"))\
        .map(lambda p: (p[0],p[1],p[2],p[3],p[4],p[5],p[6],int(p[7]),int(p[8]),int(p[9]),int(p[10]),
                        float(p[11]),float(p[12]),p[13],p[14],p[15],p[16],p[17],p[18],float(p[19]),

float(p[20]),float(p[21]),float(p[22]),float(p[23]),float(p[24]),int(p[25]),int(p[26])))

# CREATE DATA FRAME
taxi_train_df = sqlContext.createDataFrame(taxi_temp, taxi_schema)

# CREATE A CLEANED DATA-FRAME BY DROPPING SOME UN-NECESSARY COLUMNS & FILTERING FOR UNDESIRED VALUES OR
OUTLIERS
taxi_df_train_cleaned =
taxi_train_df.drop('medallion').drop('hack_license').drop('store_and_fwd_flag').drop('pickup_datetime')\
    .drop('dropoff_datetime').drop('pickup_longitude').drop('pickup_latitude').drop('dropoff_latitude')\
    .drop('dropoff_longitude').drop('tip_class').drop('total_amount').drop('tolls_amount').drop('mta_tax')\
    .drop('direct_distance').drop('surcharge')\
    .filter("passenger_count > 0 and passenger_count < 8 AND payment_type in ('CSH', 'CRD') AND tip_amount
>= 0 AND tip_amount < 30 AND fare_amount >= 1 AND fare_amount < 150 AND trip_distance > 0 AND trip_distance
< 100 AND trip_time_in_secs > 30 AND trip_time_in_secs < 7200" )

# CACHE DATA-FRAME IN MEMORY & MATERIALIZE DF IN MEMORY
taxi_df_train_cleaned.cache()
taxi_df_train_cleaned.count()

# REGISTER DATA-FRAME AS A TEMP-TABLE IN SQL-CONTEXT
taxi_df_train_cleaned.registerTempTable("taxi_train")

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 51.72 seconds

Explore the data

Once the data has been brought into Spark, the next step in the data science process is to gain deeper understanding of the data through exploration and visualization. In this section, we examine the taxi data using SQL queries and plot the target variables and prospective features for visual inspection. Specifically, we plot the frequency of passenger counts in taxi trips, the frequency of tip amounts, and how tips vary by payment amount and type.

Plot a histogram of passenger count frequencies in the sample of taxi trips

This code and subsequent snippets use SQL magic to query the sample and local magic to plot the data.

- **SQL magic (`%%sql`)** The HDInsight PySpark kernel supports easy inline HiveQL queries against the sqlContext. The (-o VARIABLE_NAME) argument persists the output of the SQL query as a Pandas DataFrame on the Jupyter server. This setting makes the output available in the local mode.
- The `%%local` **magic** is used to run code locally on the Jupyter server, which is the headnode of the HDInsight cluster. Typically, you use `%%local` magic in conjunction with the `%%sql` magic with -o parameter. The -o parameter would persist the output of the SQL query locally and then %%local magic would trigger the next set of code snippet to run locally against the output of the SQL queries that is persisted locally

The output is automatically visualized after you run the code.

This query retrieves the trips by passenger count.

```
# PLOT FREQUENCY OF PASSENGER COUNTS IN TAXI TRIPS

# HIVEQL QUERY AGAINST THE sqlContext
%%sql -q -o sqlResults
SELECT passenger_count, COUNT(*) as trip_counts
FROM taxi_train
WHERE passenger_count > 0 and passenger_count < 7
GROUP BY passenger_count
```

This code creates a local data-frame from the query output and plots the data. The `%%local` magic creates a local data-frame, `sqlResults`, which can be used for plotting with matplotlib.

> **NOTE**
> This PySpark magic is used multiple times in this walkthrough. If the amount of data is large, you should sample to create a data-frame that can fit in local memory.

```
#CREATE LOCAL DATA-FRAME AND USE FOR MATPLOTLIB PLOTTING

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER
%%local

# USE THE JUPYTER AUTO-PLOTTING FEATURE TO CREATE INTERACTIVE FIGURES.
# CLICK ON THE TYPE OF PLOT TO BE GENERATED (E.G. LINE, AREA, BAR ETC.)
sqlResults
```

Here is the code to plot the trips by passenger counts

```
# PLOT PASSENGER NUMBER VS. TRIP COUNTS
%%local
import matplotlib.pyplot as plt
%matplotlib inline

x_labels = sqlResults['passenger_count'].values
fig = sqlResults[['trip_counts']].plot(kind='bar', facecolor='lightblue')
fig.set_xticklabels(x_labels)
fig.set_title('Counts of trips by passenger count')
fig.set_xlabel('Passenger count in trips')
fig.set_ylabel('Trip counts')
plt.show()
```

OUTPUT:

You can select among several different types of visualizations (Table, Pie, Line, Area, or Bar) by using the **Type** menu buttons in the notebook. The Bar plot is shown here.

Plot a histogram of tip amounts and how tip amount varies by passenger count and fare amounts.

Use a SQL query to sample data.

```
# PLOT HISTOGRAM OF TIP AMOUNTS AND VARIATION BY PASSENGER COUNT AND PAYMENT TYPE

# HIVEQL QUERY AGAINST THE sqlContext
%%sql -q -o sqlResults
SELECT fare_amount, passenger_count, tip_amount, tipped
FROM taxi_train
WHERE passenger_count > 0
AND passenger_count < 7
AND fare_amount > 0
AND fare_amount < 200
AND payment_type in ('CSH', 'CRD')
AND tip_amount > 0
AND tip_amount < 25
```

This code cell uses the SQL query to create three plots the data.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER
%%local

# HISTOGRAM OF TIP AMOUNTS AND PASSENGER COUNT
ax1 = sqlResults[['tip_amount']].plot(kind='hist', bins=25, facecolor='lightblue')
ax1.set_title('Tip amount distribution')
ax1.set_xlabel('Tip Amount ($)')
ax1.set_ylabel('Counts')
plt.suptitle('')
plt.show()

# TIP BY PASSENGER COUNT
ax2 = sqlResults.boxplot(column=['tip_amount'], by=['passenger_count'])
ax2.set_title('Tip amount by Passenger count')
ax2.set_xlabel('Passenger count')
ax2.set_ylabel('Tip Amount ($)')
plt.suptitle('')
plt.show()

# TIP AMOUNT BY FARE AMOUNT, POINTS ARE SCALED BY PASSENGER COUNT
ax = sqlResults.plot(kind='scatter', x= 'fare_amount', y = 'tip_amount', c='blue', alpha = 0.10, s=5*
(sqlResults.passenger_count))
ax.set_title('Tip amount by Fare amount')
ax.set_xlabel('Fare Amount ($)')
ax.set_ylabel('Tip Amount ($)')
plt.axis([-2, 100, -2, 20])
plt.show()
```

OUTPUT:

Prepare the data

This section describes and provides the code for procedures used to prepare data for use in ML modeling. It shows how to do the following tasks:

- Create a new feature by binning hours into traffic time buckets
- Index and encode categorical features
- Create labeled point objects for input into ML functions
- Create a random subsampling of the data and split it into training and testing sets
- Feature scaling
- Cache objects in memory

Create a new feature by binning hours into traffic time buckets

This code shows how to create a new feature by binning hours into traffic time buckets and then how to cache the resulting data frame in memory. Where Resilient Distributed Datasets (RDDs) and data-frames are used repeatedly, caching leads to improved execution times. Accordingly, we cache RDDs and data-frames at several stages in the walkthrough.

```
# CREATE FOUR BUCKETS FOR TRAFFIC TIMES
sqlStatement = """
    SELECT *,
    CASE
     WHEN (pickup_hour <= 6 OR pickup_hour >= 20) THEN "Night"
     WHEN (pickup_hour >= 7 AND pickup_hour <= 10) THEN "AMRush"
     WHEN (pickup_hour >= 11 AND pickup_hour <= 15) THEN "Afternoon"
     WHEN (pickup_hour >= 16 AND pickup_hour <= 19) THEN "PMRush"
    END as TrafficTimeBins
    FROM taxi_train
"""
taxi_df_train_with_newFeatures = sqlContext.sql(sqlStatement)

# CACHE DATA-FRAME IN MEMORY & MATERIALIZE DF IN MEMORY
# THE .COUNT() GOES THROUGH THE ENTIRE DATA-FRAME,
# MATERIALIZES IT IN MEMORY, AND GIVES THE COUNT OF ROWS.
taxi_df_train_with_newFeatures.cache()
taxi_df_train_with_newFeatures.count()
```

OUTPUT:

126050

Index and encode categorical features for input into modeling functions

This section shows how to index or encode categorical features for input into the modeling functions. The modeling and predict functions of MLlib require features with categorical input data to be indexed or encoded

prior to use. Depending on the model, you need to index or encode them in different ways:

- **Tree-based modeling** requires categories to be encoded as numerical values (for example, a feature with three categories may be encoded with 0, 1, 2). This algorithm is provided by MLlib's StringIndexer function. This function encodes a string column of labels to a column of label indices that are ordered by label frequencies. Although indexed with numerical values for input and data handling, the tree-based algorithms can be specified to treat them appropriately as categories.
- **Logistic and Linear Regression models** require one-hot encoding, where, for example, a feature with three categories can be expanded into three feature columns, with each containing 0 or 1 depending on the category of an observation. MLlib provides OneHotEncoder function to do one-hot encoding. This encoder maps a column of label indices to a column of binary vectors, with at most a single one-value. This encoding allows algorithms that expect numerical valued features, such as logistic regression, to be applied to categorical features.

Here is the code to index and encode categorical features:

```
# INDEX AND ENCODE CATEGORICAL FEATURES

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler, VectorIndexer

# INDEX AND ENCODE VENDOR_ID
stringIndexer = StringIndexer(inputCol="vendor_id", outputCol="vendorIndex")
model = stringIndexer.fit(taxi_df_train_with_newFeatures) # Input data-frame is the cleaned one from above
indexed = model.transform(taxi_df_train_with_newFeatures)
encoder = OneHotEncoder(dropLast=False, inputCol="vendorIndex", outputCol="vendorVec")
encoded1 = encoder.transform(indexed)

# INDEX AND ENCODE RATE_CODE
stringIndexer = StringIndexer(inputCol="rate_code", outputCol="rateIndex")
model = stringIndexer.fit(encoded1)
indexed = model.transform(encoded1)
encoder = OneHotEncoder(dropLast=False, inputCol="rateIndex", outputCol="rateVec")
encoded2 = encoder.transform(indexed)

# INDEX AND ENCODE PAYMENT_TYPE
stringIndexer = StringIndexer(inputCol="payment_type", outputCol="paymentIndex")
model = stringIndexer.fit(encoded2)
indexed = model.transform(encoded2)
encoder = OneHotEncoder(dropLast=False, inputCol="paymentIndex", outputCol="paymentVec")
encoded3 = encoder.transform(indexed)

# INDEX AND TRAFFIC TIME BINS
stringIndexer = StringIndexer(inputCol="TrafficTimeBins", outputCol="TrafficTimeBinsIndex")
model = stringIndexer.fit(encoded3)
indexed = model.transform(encoded3)
encoder = OneHotEncoder(dropLast=False, inputCol="TrafficTimeBinsIndex", outputCol="TrafficTimeBinsVec")
encodedFinal = encoder.transform(indexed)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 1.28 seconds

Create labeled point objects for input into ML functions

This section contains code that shows how to index categorical text data as a labeled point data type and encode it so that it can be used to train and test MLlib logistic regression and other classification models. Labeled point objects are Resilient Distributed Datasets (RDD) formatted in a way that is needed as input data by most of ML algorithms in MLlib. A labeled point is a local vector, either dense or sparse, associated with a label/response.

This section contains code that shows how to index categorical text data as a labeled point data type and encode it so that it can be used to train and test MLlib logistic regression and other classification models. Labeled point objects are Resilient Distributed Datasets (RDD) consisting of a label (target/response variable) and feature vector. This format is needed as input by many ML algorithms in MLlib.

Here is the code to index and encode text features for binary classification.

```
# FUNCTIONS FOR BINARY CLASSIFICATION

# LOAD LIBRARIES
from pyspark.mllib.regression import LabeledPoint
from numpy import array

# INDEXING CATEGORICAL TEXT FEATURES FOR INPUT INTO TREE-BASED MODELS
def parseRowIndexingBinary(line):
    features = np.array([line.paymentIndex, line.vendorIndex, line.rateIndex, line.TrafficTimeBinsIndex,
                         line.pickup_hour, line.weekday, line.passenger_count, line.trip_time_in_secs,
                         line.trip_distance, line.fare_amount])
    labPt = LabeledPoint(line.tipped, features)
    return  labPt

# ONE-HOT ENCODING OF CATEGORICAL TEXT FEATURES FOR INPUT INTO LOGISTIC REGRESSION MODELS
def parseRowOneHotBinary(line):
    features = np.concatenate((np.array([line.pickup_hour, line.weekday, line.passenger_count,
                                         line.trip_time_in_secs, line.trip_distance, line.fare_amount]),
                                         line.vendorVec.toArray(), line.rateVec.toArray(),
                                         line.paymentVec.toArray(), line.TrafficTimeBinsVec.toArray()),
axis=0)
    labPt = LabeledPoint(line.tipped, features)
    return  labPt
```

Here is the code to encode and index categorical text features for linear regression analysis.

```
# FUNCTIONS FOR REGRESSION WITH TIP AMOUNT AS TARGET VARIABLE

# ONE-HOT ENCODING OF CATEGORICAL TEXT FEATURES FOR INPUT INTO TREE-BASED MODELS
def parseRowIndexingRegression(line):
    features = np.array([line.paymentIndex, line.vendorIndex, line.rateIndex, line.TrafficTimeBinsIndex,
                         line.pickup_hour, line.weekday, line.passenger_count, line.trip_time_in_secs,
                         line.trip_distance, line.fare_amount])

    labPt = LabeledPoint(line.tip_amount, features)
    return  labPt

# INDEXING CATEGORICAL TEXT FEATURES FOR INPUT INTO LINEAR REGRESSION MODELS
def parseRowOneHotRegression(line):
    features = np.concatenate((np.array([line.pickup_hour, line.weekday, line.passenger_count,
                                         line.trip_time_in_secs, line.trip_distance, line.fare_amount]),
                                         line.vendorVec.toArray(), line.rateVec.toArray(),
                                         line.paymentVec.toArray(), line.TrafficTimeBinsVec.toArray()),
axis=0)
    labPt = LabeledPoint(line.tip_amount, features)
    return  labPt
```

Create a random subsampling of the data and split it into training and testing sets

This code creates a random sampling of the data (25% is used here). Although it is not required for this example due to the size of the dataset, we demonstrate how you can sample here so you know how to use it for your

own problem when needed. When samples are large, sampling can save significant time while training models. Next we split the sample into a training part (75% here) and a testing part (25% here) to use in classification and regression modeling.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.sql.functions import rand

# SPECIFY SAMPLING AND SPLITTING FRACTIONS
samplingFraction = 0.25;
trainingFraction = 0.75; testingFraction = (1-trainingFraction);
seed = 1234;
encodedFinalSampled = encodedFinal.sample(False, samplingFraction, seed=seed)

# SPLIT SAMPLED DATA-FRAME INTO TRAIN/TEST
# INCLUDE RAND COLUMN FOR CREATING CROSS-VALIDATION FOLDS (FOR USE LATER IN AN ADVANCED TOPIC)
dfTmpRand = encodedFinalSampled.select("*", rand(0).alias("rand"));
trainData, testData = dfTmpRand.randomSplit([trainingFraction, testingFraction], seed=seed);

# FOR BINARY CLASSIFICATION TRAINING AND TESTING
indexedTRAINbinary = trainData.map(parseRowIndexingBinary)
indexedTESTbinary = testData.map(parseRowIndexingBinary)
oneHotTRAINbinary = trainData.map(parseRowOneHotBinary)
oneHotTESTbinary = testData.map(parseRowOneHotBinary)

# FOR REGRESSION TRAINING AND TESTING
indexedTRAINreg = trainData.map(parseRowIndexingRegression)
indexedTESTreg = testData.map(parseRowIndexingRegression)
oneHotTRAINreg = trainData.map(parseRowOneHotRegression)
oneHotTESTreg = testData.map(parseRowOneHotRegression)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 0.24 second

Feature scaling

Feature scaling, also known as data normalization, insures that features with widely disbursed values are not given excessive weigh in the objective function. The code for feature scaling uses the StandardScaler to scale the features to unit variance. It is provided by MLlib for use in linear regression with Stochastic Gradient Descent (SGD), a popular algorithm for training a wide range of other machine learning models such as regularized regressions or support vector machines (SVM).

> **NOTE**
> We have found the LinearRegressionWithSGD algorithm to be sensitive to feature scaling.

Here is the code to scale variables for use with the regularized linear SGD algorithm.

```
# FEATURE SCALING

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.feature import StandardScaler, StandardScalerModel
from pyspark.mllib.util import MLUtils

# SCALE VARIABLES FOR REGULARIZED LINEAR SGD ALGORITHM
label = oneHotTRAINreg.map(lambda x: x.label)
features = oneHotTRAINreg.map(lambda x: x.features)
scaler = StandardScaler(withMean=False, withStd=True).fit(features)
dataTMP = label.zip(scaler.transform(features.map(lambda x: Vectors.dense(x.toArray()))))
oneHotTRAINregScaled = dataTMP.map(lambda x: LabeledPoint(x[0], x[1]))

label = oneHotTESTreg.map(lambda x: x.label)
features = oneHotTESTreg.map(lambda x: x.features)
scaler = StandardScaler(withMean=False, withStd=True).fit(features)
dataTMP = label.zip(scaler.transform(features.map(lambda x: Vectors.dense(x.toArray()))))
oneHotTESTregScaled = dataTMP.map(lambda x: LabeledPoint(x[0], x[1]))

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 13.17 seconds

Cache objects in memory

The time taken for training and testing of ML algorithms can be reduced by caching the input data frame objects used for classification, regression, and scaled features.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# FOR BINARY CLASSIFICATION TRAINING AND TESTING
indexedTRAINbinary.cache()
indexedTESTbinary.cache()
oneHotTRAINbinary.cache()
oneHotTESTbinary.cache()

# FOR REGRESSION TRAINING AND TESTING
indexedTRAINreg.cache()
indexedTESTreg.cache()
oneHotTRAINreg.cache()
oneHotTESTreg.cache()

# SCALED FEATURES
oneHotTRAINregScaled.cache()
oneHotTESTregScaled.cache()

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 0.15 second

Train a binary classification model

This section shows how use three models for the binary classification task of predicting whether or not a tip is paid for a taxi trip. The models presented are:

- Regularized logistic regression
- Random forest model
- Gradient Boosting Trees

Each model building code section is split into steps:

1. **Model training** data with one parameter set
2. **Model evaluation** on a test data set with metrics
3. **Saving model** in blob for future consumption

Classification using logistic regression

The code in this section shows how to train, evaluate, and save a logistic regression model with LBFGS that predicts whether or not a tip is paid for a trip in the NYC taxi trip and fare dataset.

Train the logistic regression model using CV and hyperparameter sweeping

```
# LOGISTIC REGRESSION CLASSIFICATION WITH CV AND HYPERPARAMETER SWEEPING

# GET ACCURACY FOR HYPERPARAMETERS BASED ON CROSS-VALIDATION IN TRAINING DATA-SET

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD LIBRARIES
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from sklearn.metrics import roc_curve,auc
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.mllib.evaluation import MulticlassMetrics

# CREATE MODEL WITH ONE SET OF PARAMETERS
logitModel = LogisticRegressionWithLBFGS.train(oneHotTRAINbinary, iterations=20, initialWeights=None,
                                               regParam=0.01, regType='l2', intercept=True, corrections=10,
                                               tolerance=0.0001, validateData=True, numClasses=2)

# PRINT COEFFICIENTS AND INTERCEPT OF THE MODEL
# NOTE: There are 20 coefficient terms for the 10 features,
#       and the different categories for features: vendorVec (2), rateVec, paymentVec (6),
TrafficTimeBinsVec (4)
print("Coefficients: " + str(logitModel.weights))
print("Intercept: " + str(logitModel.intercept))

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Coefficients: [0.0082065285375, -0.0223675576104, -0.0183812028036, -3.48124578069e-05, -0.00247646947233, -0.00165897881503, 0.0675394837328, -0.111823113101, -0.324609912762, -0.204549780032, -1.36499216354, 0.591088507921, -0.664263411392, -1.00439726852, 3.46567827545, -3.51025855172, -0.0471341112232, -0.043521833294, 0.000243375810385, 0.054518719222]

Intercept: -0.0111216486893

Time taken to execute above cell: 14.43 seconds

Evaluate the binary classification model with standard metrics

```
#EVALUATE LOGISTIC REGRESSION MODEL WITH LBFGS

# RECORD START TIME
timestart = datetime.datetime.now()

# PREDICT ON TEST DATA WITH MODEL
predictionAndLabels = oneHotTESTbinary.map(lambda lp: (float(logitModel.predict(lp.features)), lp.label))

# INSTANTIATE METRICS OBJECT
metrics = BinaryClassificationMetrics(predictionAndLabels)

# AREA UNDER PRECISION-RECALL CURVE
print("Area under PR = %s" % metrics.areaUnderPR)

# AREA UNDER ROC CURVE
print("Area under ROC = %s" % metrics.areaUnderROC)
metrics = MulticlassMetrics(predictionAndLabels)

# OVERALL STATISTICS
precision = metrics.precision()
recall = metrics.recall()
f1Score = metrics.fMeasure()
print("Summary Stats")
print("Precision = %s" % precision)
print("Recall = %s" % recall)
print("F1 Score = %s" % f1Score)

## SAVE MODEL WITH DATE-STAMP
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
logisticregressionfilename = "LogisticRegressionWithLBFGS_" + datestamp;
dirfilename = modelDir + logisticregressionfilename;
logitModel.save(sc, dirfilename);

# OUTPUT PROBABILITIES AND REGISTER TEMP TABLE
logitModel.clearThreshold(); # This clears threshold for classification (0.5) and outputs probabilities
predictionAndLabelsDF = predictionAndLabels.toDF()
predictionAndLabelsDF.registerTempTable("tmp_results");

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Area under PR = 0.985297691373

Area under ROC = 0.983714670256

Summary Stats

Precision = 0.984304060189

Recall = 0.984304060189

F1 Score = 0.984304060189

Time taken to execute above cell: 57.61 seconds

Plot the ROC curve.

The *predictionAndLabelsDF* is registered as a table, *tmp_results*, in the previous cell. *tmp_results* can be used to

do queries and output results into the sqlResults data-frame for plotting. Here is the code.

```
# QUERY RESULTS
%%sql -q -o sqlResults
SELECT * from tmp_results
```

Here is the code to make predictions and plot the ROC-curve.

```
# MAKE PREDICTIONS AND PLOT ROC-CURVE

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
%matplotlib inline
from sklearn.metrics import roc_curve,auc

# MAKE PREDICTIONS
predictions_pddf = test_predictions.rename(columns={'_1': 'probability', '_2': 'label'})
prob = predictions_pddf["probability"]
fpr, tpr, thresholds = roc_curve(predictions_pddf['label'], prob, pos_label=1);
roc_auc = auc(fpr, tpr)

# PLOT ROC CURVE
plt.figure(figsize=(5,5))
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc="lower right")
plt.show()
```

OUTPUT:

Random forest classification

The code in this section shows how to train, evaluate, and save a random forest model that predicts whether or not a tip is paid for a trip in the NYC taxi trip and fare dataset.

```
#PREDICT WHETHER A TIP IS PAID OR NOT USING RANDOM FOREST

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.mllib.evaluation import MulticlassMetrics

# SPECIFY NUMBER OF CATEGORIES FOR CATEGORICAL FEATURES. FEATURE #0 HAS 2 CATEGORIES, FEATURE #2 HAS 2
CATEGORIES, AND SO ON
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}

# TRAIN RANDOMFOREST MODEL
rfModel = RandomForest.trainClassifier(indexedTRAINbinary, numClasses=2,
                                       categoricalFeaturesInfo=categoricalFeaturesInfo,
                                       numTrees=25, featureSubsetStrategy="auto",
                                       impurity='gini', maxDepth=5, maxBins=32)
## UN-COMMENT IF YOU WANT TO PRINT TREES
#print('Learned classification forest model:')
#print(rfModel.toDebugString())

# PREDICT ON TEST DATA AND EVALUATE
predictions = rfModel.predict(indexedTESTbinary.map(lambda x: x.features))
predictionAndLabels = indexedTESTbinary.map(lambda lp: lp.label).zip(predictions)

# AREA UNDER ROC CURVE
metrics = BinaryClassificationMetrics(predictionAndLabels)
print("Area under ROC = %s" % metrics.areaUnderROC)

# PERSIST MODEL IN BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
rfclassificationfilename = "RandomForestClassification_" + datestamp;
dirfilename = modelDir + rfclassificationfilename;

rfModel.save(sc, dirfilename);

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Area under ROC = 0.985297691373

Time taken to execute above cell: 31.09 seconds

Gradient boosting trees classification

The code in this section shows how to train, evaluate, and save a gradient boosting trees model that predicts whether or not a tip is paid for a trip in the NYC taxi trip and fare dataset.

```
#PREDICT WHETHER A TIP IS PAID OR NOT USING GRADIENT BOOSTING TREES

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel

# SPECIFY NUMBER OF CATEGORIES FOR CATEGORICAL FEATURES. FEATURE #0 HAS 2 CATEGORIES, FEATURE #2 HAS 2
CATEGORIES, AND SO ON
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}

gbtModel = GradientBoostedTrees.trainClassifier(indexedTRAINbinary,
categoricalFeaturesInfo=categoricalFeaturesInfo, numIterations=5)
## UNCOMMENT IF YOU WANT TO PRINT TREE DETAILS
#print('Learned classification GBT model:')
#print(bgtModel.toDebugString())

# PREDICT ON TEST DATA AND EVALUATE
predictions = gbtModel.predict(indexedTESTbinary.map(lambda x: x.features))
predictionAndLabels = indexedTESTbinary.map(lambda lp: lp.label).zip(predictions)

# AREA UNDER ROC CURVE
metrics = BinaryClassificationMetrics(predictionAndLabels)
print("Area under ROC = %s" % metrics.areaUnderROC)

# PERSIST MODEL IN A BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
btclassificationfilename = "GradientBoostingTreeClassification_" + datestamp;
dirfilename = modelDir + btclassificationfilename;

gbtModel.save(sc, dirfilename)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Area under ROC = 0.985297691373

Time taken to execute above cell: 19.76 seconds

Train a regression model

This section shows how use three models for the regression task of predicting the amount of the tip paid for a taxi trip based on other tip features. The models presented are:

- Regularized linear regression
- Random forest
- Gradient Boosting Trees

These models were described in the introduction. Each model building code section is split into steps:

1. **Model training** data with one parameter set
2. **Model evaluation** on a test data set with metrics
3. **Saving model** in blob for future consumption

Linear regression with SGD

The code in this section shows how to use scaled features to train a linear regression that uses stochastic gradient descent (SGD) for optimization, and how to score, evaluate, and save the model in Azure Blob Storage

(WASB).

TIP

In our experience, there can be issues with the convergence of LinearRegressionWithSGD models, and parameters need to be changed/optimized carefully for obtaining a valid model. Scaling of variables significantly helps with convergence.

```
#PREDICT TIP AMOUNTS USING LINEAR REGRESSION WITH SGD

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD LIBRARIES
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD, LinearRegressionModel
from pyspark.mllib.evaluation import RegressionMetrics
from scipy import stats

# USE SCALED FEATURES TO TRAIN MODEL
linearModel = LinearRegressionWithSGD.train(oneHotTRAINregScaled, iterations=100, step = 0.1, regType='l2',
regParam=0.1, intercept = True)

# PRINT COEFFICIENTS AND INTERCEPT OF THE MODEL
# NOTE: There are 20 coefficient terms for the 10 features,
#       and the different categories for features: vendorVec (2), rateVec, paymentVec (6),
TrafficTimeBinsVec (4)
print("Coefficients: " + str(linearModel.weights))
print("Intercept: " + str(linearModel.intercept))

# SCORE ON SCALED TEST DATA-SET & EVALUATE
predictionAndLabels = oneHotTESTregScaled.map(lambda lp: (float(linearModel.predict(lp.features)),
lp.label))
testMetrics = RegressionMetrics(predictionAndLabels)

# PRINT TEST METRICS
print("RMSE = %s" % testMetrics.rootMeanSquaredError)
print("R-sqr = %s" % testMetrics.r2)

# SAVE MODEL WITH DATE-STAMP IN THE DEFAULT BLOB FOR THE CLUSTER
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
linearregressionfilename = "LinearRegressionWithSGD_" + datestamp;
dirfilename = modelDir + linearregressionfilename;

linearModel.save(sc, dirfilename)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Coefficients: [0.00457675809917, -0.0226314167349, -0.0191910355236, 0.246793409578, 0.312047890459, 0.359634405999, 0.00928692253981, -0.000987181489428, -0.0888306617845, 0.0569376211553, 0.115519551711, 0.149250164995, -0.00990211159703, -0.00637410344522, 0.545083566179, -0.536756072402, 0.0105762393099, -0.0130117577055, 0.0129304737772, -0.00171065945959]

Intercept: 0.853872718283

RMSE = 1.24190115863

R-sqr = 0.608017146081

Time taken to execute above cell: 58.42 seconds

Random Forest regression

The code in this section shows how to train, evaluate, and save a random forest regression that predicts tip amount for the NYC taxi trip data.

```
#PREDICT TIP AMOUNTS USING RANDOM FOREST

# RECORD START TIME
timestart= datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils
from pyspark.mllib.evaluation import RegressionMetrics

## TRAIN MODEL
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}
rfModel = RandomForest.trainRegressor(indexedTRAINreg, categoricalFeaturesInfo=categoricalFeaturesInfo,
                                      numTrees=25, featureSubsetStrategy="auto",
                                      impurity='variance', maxDepth=10, maxBins=32)
## UN-COMMENT IF YOU WANT TO PRING TREES
#print('Learned classification forest model:')
#print(rfModel.toDebugString())

## PREDICT AND EVALUATE ON TEST DATA-SET
predictions = rfModel.predict(indexedTESTreg.map(lambda x: x.features))
predictionAndLabels = oneHotTESTreg.map(lambda lp: lp.label).zip(predictions)

# TEST METRICS
testMetrics = RegressionMetrics(predictionAndLabels)
print("RMSE = %s" % testMetrics.rootMeanSquaredError)
print("R-sqr = %s" % testMetrics.r2)

# SAVE MODEL IN BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
rfregressionfilename = "RandomForestRegression_" + datestamp;
dirfilename = modelDir + rfregressionfilename;

rfModel.save(sc, dirfilename);

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

RMSE = 0.891209218139

R-sqr = 0.759661334921

Time taken to execute above cell: 49.21 seconds

Gradient boosting trees regression

The code in this section shows how to train, evaluate, and save a gradient boosting trees model that predicts tip amount for the NYC taxi trip data.

Train and evaluate

```
#PREDICT TIP AMOUNTS USING GRADIENT BOOSTING TREES

# RECORD START TIME
timestart= datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel
from pyspark.mllib.util import MLUtils

## TRAIN MODEL
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}
gbtModel = GradientBoostedTrees.trainRegressor(indexedTRAINreg,
categoricalFeaturesInfo=categoricalFeaturesInfo,
                                                numIterations=10, maxBins=32, maxDepth = 4,
learningRate=0.1)

## EVALUATE A TEST DATA-SET
predictions = gbtModel.predict(indexedTESTreg.map(lambda x: x.features))
predictionAndLabels = indexedTESTreg.map(lambda lp: lp.label).zip(predictions)

# TEST METRICS
testMetrics = RegressionMetrics(predictionAndLabels)
print("RMSE = %s" % testMetrics.rootMeanSquaredError)
print("R-sqr = %s" % testMetrics.r2)

# SAVE MODEL IN BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
btregressionfilename = "GradientBoostingTreeRegression_" + datestamp;
dirfilename = modelDir + btregressionfilename;
gbtModel.save(sc, dirfilename)

# CONVERT RESULTS TO DF AND REGISTER TEMP TABLE
test_predictions = sqlContext.createDataFrame(predictionAndLabels)
test_predictions.registerTempTable("tmp_results");

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

RMSE = 0.908473148639

R-sqr = 0.753835096681

Time taken to execute above cell: 34.52 seconds

Plot

tmp_results is registered as a Hive table in the previous cell. Results from the table are output into the *sqlResults* data-frame for plotting. Here is the code

```
# PLOT SCATTER-PLOT BETWEEN ACTUAL AND PREDICTED TIP VALUES

# SELECT RESULTS
%%sql -q -o sqlResults
SELECT * from tmp_results
```

Here is the code to plot the data using the Jupyter server.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
%matplotlib inline
import numpy as np

# PLOT
ax = test_predictions_pddf.plot(kind='scatter', figsize = (6,6), x='_1', y='_2', color='blue', alpha = 0.25,
label='Actual vs. predicted');
fit = np.polyfit(test_predictions_pddf['_1'], test_predictions_pddf['_2'], deg=1)
ax.set_title('Actual vs. Predicted Tip Amounts ($)')
ax.set_xlabel("Actual")
ax.set_ylabel("Predicted")
ax.plot(test_predictions_pddf['_1'], fit[0] * test_predictions_pddf['_1'] + fit[1], color='magenta')
plt.axis([-1, 20, -1, 20])
plt.show(ax)
```

OUTPUT:

Clean up objects from memory

Use `unpersist()` to delete objects cached in memory.

```
# REMOVE ORIGINAL DFs
taxi_df_train_cleaned.unpersist()
taxi_df_train_with_newFeatures.unpersist()

# FOR BINARY CLASSIFICATION TRAINING AND TESTING
indexedTRAINbinary.unpersist()
indexedTESTbinary.unpersist()
oneHotTRAINbinary.unpersist()
oneHotTESTbinary.unpersist()

# FOR REGRESSION TRAINING AND TESTING
indexedTRAINreg.unpersist()
indexedTESTreg.unpersist()
oneHotTRAINreg.unpersist()
oneHotTESTreg.unpersist()

# SCALED FEATURES
oneHotTRAINregScaled.unpersist()
oneHotTESTregScaled.unpersist()
```

Save the models

To consume and score an independent dataset described in the Score and evaluate Spark-built machine learning models topic, you need to copy and paste these file names containing the saved models created here into the Consumption Jupyter notebook. Here is the code to print out the paths to model files you need there.

```
# MODEL FILE LOCATIONS FOR CONSUMPTION
print "logisticRegFileLoc = modelDir + \"" + logisticregressionfilename + "\"";
print "linearRegFileLoc = modelDir + \"" + linearregressionfilename + "\"";
print "randomForestClassificationFileLoc = modelDir + \"" + rfclassificationfilename + "\"";
print "randomForestRegFileLoc = modelDir + \"" + rfregressionfilename + "\"";
print "BoostedTreeClassificationFileLoc = modelDir + \"" + btclassificationfilename + "\"";
print "BoostedTreeRegressionFileLoc = modelDir + \"" + btregressionfilename + "\"";
```

OUTPUT

logisticRegFileLoc = modelDir + "LogisticRegressionWithLBFGS_2016-05-0317_03_23.516568"

linearRegFileLoc = modelDir + "LinearRegressionWithSGD_2016-05-0317_05_21.577773"

randomForestClassificationFileLoc = modelDir + "RandomForestClassification_2016-05-0317_04_11.950206"

randomForestRegFileLoc = modelDir + "RandomForestRegression_2016-05-0317_06_08.723736"

BoostedTreeClassificationFileLoc = modelDir + "GradientBoostingTreeClassification_2016-05-0317_04_36.346583"

BoostedTreeRegressionFileLoc = modelDir + "GradientBoostingTreeRegression_2016-05-0317_06_51.737282"

What's next?

Now that you have created regression and classification models with the Spark MlLib, you are ready to learn how to score and evaluate these models. The advanced data exploration and modeling notebook dives deeper into including cross-validation, hyper-parameter sweeping, and model evaluation.

Model consumption: To learn how to score and evaluate the classification and regression models created in this topic, see Score and evaluate Spark-built machine learning models.

Cross-validation and hyperparameter sweeping: See Advanced data exploration and modeling with Spark on how models can be trained using cross-validation and hyper-parameter sweeping

Advanced data exploration and modeling with Spark

11/19/2021 • 37 minutes to read • Edit Online

This walkthrough uses HDInsight Spark to do data exploration and train binary classification and regression models using cross-validation and hyperparameter optimization on a sample of the NYC taxi trip and fare 2013 dataset. It walks you through the steps of the Data Science Process, end-to-end, using an HDInsight Spark cluster for processing and Azure blobs to store the data and the models. The process explores and visualizes data brought in from an Azure Storage Blob and then prepares the data to build predictive models. Python has been used to code the solution and to show the relevant plots. These models are build using the Spark MLlib toolkit to do binary classification and regression modeling tasks.

- The **binary classification** task is to predict whether or not a tip is paid for the trip.
- The **regression** task is to predict the amount of the tip based on other tip features.

The modeling steps also contain code showing how to train, evaluate, and save each type of model. The topic covers some of the same ground as the Data exploration and modeling with Spark topic. But it is more "advanced" in that it also uses cross-validation with hyperparameter sweeping to train optimally accurate classification and regression models.

Cross-validation (CV) is a technique that assesses how well a model trained on a known set of data generalizes to predicting the features of datasets on which it has not been trained. A common implementation used here is to divide a dataset into K folds and then train the model in a round-robin fashion on all but one of the folds. The ability of the model to prediction accurately when tested against the independent dataset in this fold not used to train the model is assessed.

Hyperparameter optimization is the problem of choosing a set of hyperparameters for a learning algorithm, usually with the goal of optimizing a measure of the algorithm's performance on an independent data set. **Hyperparameters** are values that must be specified outside of the model training procedure. Assumptions about these values can impact the flexibility and accuracy of the models. Decision trees have hyperparameters, for example, such as the desired depth and number of leaves in the tree. Support Vector Machines (SVMs) require setting a misclassification penalty term.

A common way to perform hyperparameter optimization used here is a grid search, or a **parameter sweep**. This search goes through a subset of the hyperparameter space for a learning algorithm. Cross validation can supply a performance metric to sort out the optimal results produced by the grid search algorithm. CV used with hyperparameter sweeping helps limit problems like overfitting a model to training data so that the model retains the capacity to apply to the general set of data from which the training data was extracted.

The models we use include logistic and linear regression, random forests, and gradient boosted trees:

- Linear regression with SGD is a linear regression model that uses a Stochastic Gradient Descent (SGD) method and for optimization and feature scaling to predict the tip amounts paid.
- Logistic regression with LBFGS or "logit" regression, is a regression model that can be used when the dependent variable is categorical to do data classification. LBFGS is a quasi-Newton optimization algorithm that approximates the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm using a limited amount of computer memory and that is widely used in machine learning.
- Random forests are ensembles of decision trees. They combine many decision trees to reduce the risk of overfitting. Random forests are used for regression and classification and can handle categorical features and can be extended to the multiclass classification setting. They do not require feature scaling and are able to

capture non-linearities and feature interactions. Random forests are one of the most successful machine learning models for classification and regression.

- Gradient boosted trees (GBTS) are ensembles of decision trees. GBTS train decision trees iteratively to minimize a loss function. GBTS is used for regression and classification and can handle categorical features, do not require feature scaling, and are able to capture non-linearities and feature interactions. They can also be used in a multiclass-classification setting.

Modeling examples using CV and Hyperparameter sweep are shown for the binary classification problem. Simpler examples (without parameter sweeps) are presented in the main topic for regression tasks. But in the appendix, validation using elastic net for linear regression and CV with parameter sweep using for random forest regression are also presented. The **elastic net** is a regularized regression method for fitting linear regression models that linearly combines the L1 and L2 metrics as penalties of the lasso and ridge methods.

> **NOTE**
> Although the Spark MLlib toolkit is designed to work on large datasets, a relatively small sample (~30 Mb using 170K rows, about 0.1% of the original NYC dataset) is used here for convenience. The exercise given here runs efficiently (in about 10 minutes) on an HDInsight cluster with 2 worker nodes. The same code, with minor modifications, can be used to process larger data-sets, with appropriate modifications for caching data in memory and changing the cluster size.

Setup: Spark clusters and notebooks

Setup steps and code are provided in this walkthrough for using an HDInsight Spark 1.6. But Jupyter notebooks are provided for both HDInsight Spark 1.6 and Spark 2.0 clusters. A description of the notebooks and links to them are provided in the Readme.md for the GitHub repository containing them. Moreover, the code here and in the linked notebooks is generic and should work on any Spark cluster. If you are not using HDInsight Spark, the cluster setup and management steps may be slightly different from what is shown here. For convenience, here are the links to the Jupyter notebooks for Spark 1.6 and 2.0 to be run in the pyspark kernel of the Jupyter Notebook server:

Spark 1.6 notebooks

pySpark-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb: Includes topics in notebook #1, and model development using hyperparameter tuning and cross-validation.

Spark 2.0 notebooks

Spark2.0-pySpark3-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb: This file provides information on how to perform data exploration, modeling, and scoring in Spark 2.0 clusters.

> **WARNING**
> Billing for HDInsight clusters is prorated per minute, whether you use them or not. Be sure to delete your cluster after you finish using it. See how to delete an HDInsight cluster.

Setup: storage locations, libraries, and the preset Spark context

Spark is able to read and write to Azure Storage Blob (also known as WASB). So any of your existing data stored there can be processed using Spark and the results stored again in WASB.

To save models or files in WASB, the path needs to be specified properly. The default container attached to the Spark cluster can be referenced using a path beginning with: "wasb:///". Other locations are referenced by "wasb://".

Set directory paths for storage locations in WASB

The following code sample specifies the location of the data to be read and the path for the model storage directory to which the model output is saved:

```
# SET PATHS TO FILE LOCATIONS: DATA AND MODEL STORAGE

# LOCATION OF TRAINING DATA
taxi_train_file_loc =
"wasb://mllibwalkthroughs@cdspsparksamples.blob.core.windows.net/Data/NYCTaxi/JoinedTaxiTripFare.Point1Pct.T
rain.tsv";

# SET THE MODEL STORAGE DIRECTORY PATH
# NOTE THAT THE FINAL BACKSLASH IN THE PATH IS NEEDED.
modelDir = "wasb:///user/remoteuser/NYCTaxi/Models/";

# PRINT START TIME
import datetime
datetime.datetime.now()
```

OUTPUT

datetime.datetime(2016, 4, 18, 17, 36, 27, 832799)

Import libraries

Import necessary libraries with the following code:

```
# LOAD PYSPARK LIBRARIES
import pyspark
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.sql import SQLContext
import matplotlib
import matplotlib.pyplot as plt
from pyspark.sql import Row
from pyspark.sql.functions import UserDefinedFunction
from pyspark.sql.types import *
import atexit
from numpy import array
import numpy as np
import datetime
```

Preset Spark context and PySpark magics

The PySpark kernels that are provided with Jupyter notebooks have a preset context. So you do not need to set the Spark or Hive contexts explicitly before you start working with the application you are developing. These contexts are available for you by default. These contexts are:

- sc - for Spark
- sqlContext - for Hive

The PySpark kernel provides some predefined "magics", which are special commands that you can call with %%. There are two such commands that are used in these code samples.

- **%%local** Specifies that the code in subsequent lines is to be executed locally. Code must be valid Python code.
- **%%sql -o <variable name>** Executes a Hive query against the sqlContext. If the -o parameter is passed, the result of the query is persisted in the %%local Python context as a Pandas DataFrame.

For more information on the kernels for Jupyter notebooks and the predefined "magics" that they provide, see Kernels available for Jupyter notebooks with HDInsight Spark Linux clusters on HDInsight.

Data ingestion from public blob:

The first step in the data science process is to ingest the data to be analyzed from sources where it resides into your data exploration and modeling environment. This environment is Spark in this walkthrough. This section contains the code to complete a series of tasks:

- ingest the data sample to be modeled
- read in the input dataset (stored as a .tsv file)
- format and clean the data
- create and cache objects (RDDs or data-frames) in memory
- register it as a temp-table in SQL-context.

Here is the code for data ingestion.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# IMPORT FILE FROM PUBLIC BLOB
taxi_train_file = sc.textFile(taxi_train_file_loc)

# GET SCHEMA OF THE FILE FROM HEADER
schema_string = taxi_train_file.first()
fields = [StructField(field_name, StringType(), True) for field_name in schema_string.split('\t')]
fields[7].dataType = IntegerType() #Pickup hour
fields[8].dataType = IntegerType() # Pickup week
fields[9].dataType = IntegerType() # Weekday
fields[10].dataType = IntegerType() # Passenger count
fields[11].dataType = FloatType() # Trip time in secs
fields[12].dataType = FloatType() # Trip distance
fields[19].dataType = FloatType() # Fare amount
fields[20].dataType = FloatType() # Surcharge
fields[21].dataType = FloatType() # Mta_tax
fields[22].dataType = FloatType() # Tip amount
fields[23].dataType = FloatType() # Tolls amount
fields[24].dataType = FloatType() # Total amount
fields[25].dataType = IntegerType() # Tipped or not
fields[26].dataType = IntegerType() # Tip class
taxi_schema = StructType(fields)

# PARSE FIELDS AND CONVERT DATA TYPE FOR SOME FIELDS
taxi_header = taxi_train_file.filter(lambda l: "medallion" in l)
taxi_temp = taxi_train_file.subtract(taxi_header).map(lambda k: k.split("\t"))\
            .map(lambda p: (p[0],p[1],p[2],p[3],p[4],p[5],p[6],int(p[7]),int(p[8]),int(p[9]),int(p[10]),
                        float(p[11]),float(p[12]),p[13],p[14],p[15],p[16],p[17],p[18],float(p[19]),

float(p[20]),float(p[21]),float(p[22]),float(p[23]),float(p[24]),int(p[25]),int(p[26])))

# CREATE DATA FRAME
taxi_train_df = sqlContext.createDataFrame(taxi_temp, taxi_schema)

# CREATE A CLEANED DATA-FRAME BY DROPPING SOME UN-NECESSARY COLUMNS & FILTERING FOR UNDESIRED VALUES OR
OUTLIERS
taxi_df_train_cleaned =
taxi_train_df.drop('medallion').drop('hack_license').drop('store_and_fwd_flag').drop('pickup_datetime')\
    .drop('dropoff_datetime').drop('pickup_longitude').drop('pickup_latitude').drop('dropoff_latitude')\
    .drop('dropoff_longitude').drop('tip_class').drop('total_amount').drop('tolls_amount').drop('mta_tax')\
    .drop('direct_distance').drop('surcharge')\
    .filter("passenger_count > 0 and passenger_count < 8 AND payment_type in ('CSH', 'CRD') AND tip_amount
>= 0 AND tip_amount < 30 AND fare_amount >= 1 AND fare_amount < 150 AND trip_distance > 0 AND trip_distance
< 100 AND trip_time_in_secs > 30 AND trip_time_in_secs < 7200" )

# CACHE & MATERIALIZE DATA-FRAME IN MEMORY. GOING THROUGH AND COUNTING NUMBER OF ROWS MATERIALIZES THE DATA-
FRAME IN MEMORY
taxi_df_train_cleaned.cache()
taxi_df_train_cleaned.count()

# REGISTER DATA-FRAME AS A TEMP-TABLE IN SQL-CONTEXT
taxi_df_train_cleaned.registerTempTable("taxi_train")

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 276.62 seconds

Data exploration & visualization

Once the data has been brought into Spark, the next step in the data science process is to gain deeper understanding of the data through exploration and visualization. In this section, we examine the taxi data using SQL queries and plot the target variables and prospective features for visual inspection. Specifically, we plot the frequency of passenger counts in taxi trips, the frequency of tip amounts, and how tips vary by payment amount and type.

Plot a histogram of passenger count frequencies in the sample of taxi trips

This code and subsequent snippets use SQL magic to query the sample and local magic to plot the data.

- **SQL magic (`%%sql`)** The HDInsight PySpark kernel supports easy inline HiveQL queries against the sqlContext. The (-o VARIABLE_NAME) argument persists the output of the SQL query as a Pandas DataFrame on the Jupyter server. This means it is available in the local mode.
- The `%%local` **magic** is used to run code locally on the Jupyter server, which is the headnode of the HDInsight cluster. Typically, you use `%%local` magic after the `%%sql -o` magic is used to run a query. The -o parameter would persist the output of the SQL query locally. Then the `%%local` magic triggers the next set of code snippets to run locally against the output of the SQL queries that has been persisted locally. The output is automatically visualized after you run the code.

This query retrieves the trips by passenger count.

```
# PLOT FREQUENCY OF PASSENGER COUNTS IN TAXI TRIPS

# SQL QUERY
%%sql -q -o sqlResults
SELECT passenger_count, COUNT(*) as trip_counts FROM taxi_train WHERE passenger_count > 0 and
passenger_count < 7 GROUP BY passenger_count
```

This code creates a local data-frame from the query output and plots the data. The `%%local` magic creates a local data-frame, `sqlResults`, which can be used for plotting with matplotlib.

> **NOTE**
> This PySpark magic is used multiple times in this walkthrough. If the amount of data is large, you should sample to create a data-frame that can fit in local memory.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER
%%local

# USE THE JUPYTER AUTO-PLOTTING FEATURE TO CREATE INTERACTIVE FIGURES.
# CLICK ON THE TYPE OF PLOT TO BE GENERATED (E.G. LINE, AREA, BAR ETC.)
sqlResults
```

Here is the code to plot the trips by passenger counts

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
import matplotlib.pyplot as plt
%matplotlib inline

# PLOT PASSENGER NUMBER VS TRIP COUNTS
x_labels = sqlResults['passenger_count'].values
fig = sqlResults[['trip_counts']].plot(kind='bar', facecolor='lightblue')
fig.set_xticklabels(x_labels)
fig.set_title('Counts of trips by passenger count')
fig.set_xlabel('Passenger count in trips')
fig.set_ylabel('Trip counts')
plt.show()
```

OUTPUT

You can select among several different types of visualizations (Table, Pie, Line, Area, or Bar) by using the **Type** menu buttons in the notebook. The Bar plot is shown here.

Plot a histogram of tip amounts and how tip amount varies by passenger count and fare amounts.

Use a SQL query to sample data..

```
# SQL SQUERY
%%sql -q -o sqlResults
    SELECT fare_amount, passenger_count, tip_amount, tipped
    FROM taxi_train
    WHERE passenger_count > 0
    AND passenger_count < 7
    AND fare_amount > 0
    AND fare_amount < 200
    AND payment_type in ('CSH', 'CRD')
    AND tip_amount > 0
    AND tip_amount < 25
```

This code cell uses the SQL query to create three plots the data.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
%matplotlib inline

# TIP BY PAYMENT TYPE AND PASSENGER COUNT
ax1 = resultsPDDF[['tip_amount']].plot(kind='hist', bins=25, facecolor='lightblue')
ax1.set_title('Tip amount distribution')
ax1.set_xlabel('Tip Amount ($)')
ax1.set_ylabel('Counts')
plt.suptitle('')
plt.show()

# TIP BY PASSENGER COUNT
ax2 = resultsPDDF.boxplot(column=['tip_amount'], by=['passenger_count'])
ax2.set_title('Tip amount ($) by Passenger count')
ax2.set_xlabel('Passenger count')
ax2.set_ylabel('Tip Amount ($)')
plt.suptitle('')
plt.show()

# TIP AMOUNT BY FARE AMOUNT, POINTS ARE SCALED BY PASSENGER COUNT
ax = resultsPDDF.plot(kind='scatter', x= 'fare_amount', y = 'tip_amount', c='blue', alpha = 0.10, s=5*
(resultsPDDF.passenger_count))
ax.set_title('Tip amount by Fare amount ($)')
ax.set_xlabel('Fare Amount')
ax.set_ylabel('Tip Amount')
plt.axis([-2, 120, -2, 30])
plt.show()
```

OUTPUT:

Feature engineering, transformation, and data preparation for modeling

This section describes and provides the code for procedures used to prepare data for use in ML modeling. It shows how to do the following tasks:

- Create a new feature by partitioning hours into traffic time bins
- Index and on-hot encode categorical features
- Create labeled point objects for input into ML functions
- Create a random subsampling of the data and split it into training and testing sets
- Feature scaling
- Cache objects in memory

Create a new feature by partitioning traffic times into bins

This code shows how to create a new feature by partitioning traffic times into bins and then how to cache the resulting data frame in memory. Caching leads to improved execution time where Resilient Distributed Datasets (RDDs) and data-frames are used repeatedly. So, we cache RDDs and data-frames at several stages in this walkthrough.

```
# CREATE FOUR BUCKETS FOR TRAFFIC TIMES
sqlStatement = """
    SELECT *,
    CASE
     WHEN (pickup_hour <= 6 OR pickup_hour >= 20) THEN "Night"
     WHEN (pickup_hour >= 7 AND pickup_hour <= 10) THEN "AMRush"
     WHEN (pickup_hour >= 11 AND pickup_hour <= 15) THEN "Afternoon"
     WHEN (pickup_hour >= 16 AND pickup_hour <= 19) THEN "PMRush"
    END as TrafficTimeBins
    FROM taxi_train
"""
taxi_df_train_with_newFeatures = sqlContext.sql(sqlStatement)
```

```
# CACHE DATA-FRAME IN MEMORY & MATERIALIZE DF IN MEMORY
# THE .COUNT() GOES THROUGH THE ENTIRE DATA-FRAME,
# MATERIALIZES IT IN MEMORY, AND GIVES THE COUNT OF ROWS.
taxi_df_train_with_newFeatures.cache()
taxi_df_train_with_newFeatures.count()
```

OUTPUT

126050

Index and one-hot encode categorical features

This section shows how to index or encode categorical features for input into the modeling functions. The modeling and predict functions of MLlib require that features with categorical input data be indexed or encoded prior to use.

Depending on the model, you need to index or encode them in different ways. For example, Logistic and Linear Regression models require one-hot encoding, where, for example, a feature with three categories can be expanded into three feature columns, with each containing 0 or 1 depending on the category of an observation. MLlib provides OneHotEncoder function to do one-hot encoding. This encoder maps a column of label indices to a column of binary vectors, with at most a single one-value. This encoding allows algorithms that expect numerical valued features, such as logistic regression, to be applied to categorical features.

Here is the code to index and encode categorical features:

```
# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler, OneHotEncoder, VectorIndexer

# INDEX AND ENCODE VENDOR_ID
stringIndexer = StringIndexer(inputCol="vendor_id", outputCol="vendorIndex")
model = stringIndexer.fit(taxi_df_train_with_newFeatures) # Input data-frame is the cleaned one from above
indexed = model.transform(taxi_df_train_with_newFeatures)
encoder = OneHotEncoder(dropLast=False, inputCol="vendorIndex", outputCol="vendorVec")
encoded1 = encoder.transform(indexed)

# INDEX AND ENCODE RATE_CODE
stringIndexer = StringIndexer(inputCol="rate_code", outputCol="rateIndex")
model = stringIndexer.fit(encoded1)
indexed = model.transform(encoded1)
encoder = OneHotEncoder(dropLast=False, inputCol="rateIndex", outputCol="rateVec")
encoded2 = encoder.transform(indexed)

# INDEX AND ENCODE PAYMENT_TYPE
stringIndexer = StringIndexer(inputCol="payment_type", outputCol="paymentIndex")
model = stringIndexer.fit(encoded2)
indexed = model.transform(encoded2)
encoder = OneHotEncoder(dropLast=False, inputCol="paymentIndex", outputCol="paymentVec")
encoded3 = encoder.transform(indexed)

# INDEX AND TRAFFIC TIME BINS
stringIndexer = StringIndexer(inputCol="TrafficTimeBins", outputCol="TrafficTimeBinsIndex")
model = stringIndexer.fit(encoded3)
indexed = model.transform(encoded3)
encoder = OneHotEncoder(dropLast=False, inputCol="TrafficTimeBinsIndex", outputCol="TrafficTimeBinsVec")
encodedFinal = encoder.transform(indexed)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 3.14 seconds

Create labeled point objects for input into ML functions

This section contains code that shows how to index categorical text data as a labeled point data type and how to encode it. This transformation prepares text data to be used to train and test MLlib logistic regression and other classification models. Labeled point objects are Resilient Distributed Datasets (RDD) formatted in a way that is needed as input data by most of ML algorithms in MLlib. A labeled point is a local vector, either dense or sparse, associated with a label/response.

Here is the code to index and encode text features for binary classification.

```
# FUNCTIONS FOR BINARY CLASSIFICATION

# LOAD LIBRARIES
from pyspark.mllib.regression import LabeledPoint
from numpy import array

# INDEXING CATEGORICAL TEXT FEATURES FOR INPUT INTO TREE-BASED MODELS
def parseRowIndexingBinary(line):
    features = np.array([line.paymentIndex, line.vendorIndex, line.rateIndex, line.pickup_hour,
line.weekday,
                         line.passenger_count, line.trip_time_in_secs, line.trip_distance,
line.fare_amount])
    labPt = LabeledPoint(line.tipped, features)
    return  labPt

# ONE-HOT ENCODING OF CATEGORICAL TEXT FEATURES FOR INPUT INTO LOGISTIC REGRESSION MODELS
def parseRowOneHotBinary(line):
    features = np.concatenate((np.array([line.pickup_hour, line.weekday, line.passenger_count,
                                         line.trip_time_in_secs, line.trip_distance, line.fare_amount]),
                                line.vendorVec.toArray(), line.rateVec.toArray(), line.paymentVec.toArray()),
axis=0)
    labPt = LabeledPoint(line.tipped, features)
    return  labPt
```

Here is the code to encode and index categorical text features for linear regression analysis.

```
# FUNCTIONS FOR REGRESSION WITH TIP AMOUNT AS TARGET VARIABLE

# ONE-HOT ENCODING OF CATEGORICAL TEXT FEATURES FOR INPUT INTO TREE-BASED MODELS
def parseRowIndexingRegression(line):
    features = np.array([line.paymentIndex, line.vendorIndex, line.rateIndex, line.TrafficTimeBinsIndex,
                         line.pickup_hour, line.weekday, line.passenger_count, line.trip_time_in_secs,
                         line.trip_distance, line.fare_amount])
    labPt = LabeledPoint(line.tip_amount, features)
    return  labPt

# INDEXING CATEGORICAL TEXT FEATURES FOR INPUT INTO LINEAR REGRESSION MODELS
def parseRowOneHotRegression(line):
    features = np.concatenate((np.array([line.pickup_hour, line.weekday, line.passenger_count,
                                         line.trip_time_in_secs, line.trip_distance, line.fare_amount]),
                                         line.vendorVec.toArray(), line.rateVec.toArray(),
                                         line.paymentVec.toArray(), line.TrafficTimeBinsVec.toArray()),
axis=0)
    labPt = LabeledPoint(line.tip_amount, features)
    return  labPt
```

Create a random subsampling of the data and split it into training and testing sets

This code creates a random sampling of the data (25% is used here). Although it is not required for this example due to the size of the dataset, we demonstrate how you can sample the data here. Then you know how to use it for your own problem if needed. When samples are large, sampling can save significant time while training models. Next we split the sample into a training part (75% here) and a testing part (25% here) to use in classification and regression modeling.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# SPECIFY SAMPLING AND SPLITTING FRACTIONS
from pyspark.sql.functions import rand

samplingFraction = 0.25;
trainingFraction = 0.75; testingFraction = (1-trainingFraction);
seed = 1234;
encodedFinalSampled = encodedFinal.sample(False, samplingFraction, seed=seed)

# SPLIT SAMPLED DATA-FRAME INTO TRAIN/TEST, WITH A RANDOM COLUMN ADDED FOR DOING CV (SHOWN LATER)
# INCLUDE RAND COLUMN FOR CREATING CROSS-VALIDATION FOLDS
dfTmpRand = encodedFinalSampled.select("*", rand(0).alias("rand"));
trainData, testData = dfTmpRand.randomSplit([trainingFraction, testingFraction], seed=seed);

# CACHE TRAIN AND TEST DATA
trainData.cache()
testData.cache()

# FOR BINARY CLASSIFICATION TRAINING AND TESTING
indexedTRAINbinary = trainData.map(parseRowIndexingBinary)
indexedTESTbinary = testData.map(parseRowIndexingBinary)
oneHotTRAINbinary = trainData.map(parseRowOneHotBinary)
oneHotTESTbinary = testData.map(parseRowOneHotBinary)

# FOR REGRESSION TRAINING AND TESTING
indexedTRAINreg = trainData.map(parseRowIndexingRegression)
indexedTESTreg = testData.map(parseRowIndexingRegression)
oneHotTRAINreg = trainData.map(parseRowOneHotRegression)
oneHotTESTreg = testData.map(parseRowOneHotRegression)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 0.31 second

Feature scaling

Feature scaling, also known as data normalization, insures that features with widely disbursed values are not given excessive weigh in the objective function. The code for feature scaling uses the StandardScaler to scale the features to unit variance. It is provided by MLlib for use in linear regression with Stochastic Gradient Descent (SGD). SGD is a popular algorithm for training a wide range of other machine learning models such as regularized regressions or support vector machines (SVM).

> **TIP**
> We have found the LinearRegressionWithSGD algorithm to be sensitive to feature scaling.

Here is the code to scale variables for use with the regularized linear SGD algorithm.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.feature import StandardScaler, StandardScalerModel
from pyspark.mllib.util import MLUtils

# SCALE VARIABLES FOR REGULARIZED LINEAR SGD ALGORITHM
label = oneHotTRAINreg.map(lambda x: x.label)
features = oneHotTRAINreg.map(lambda x: x.features)
scaler = StandardScaler(withMean=False, withStd=True).fit(features)
dataTMP = label.zip(scaler.transform(features.map(lambda x: Vectors.dense(x.toArray()))))
oneHotTRAINregScaled = dataTMP.map(lambda x: LabeledPoint(x[0], x[1]))

label = oneHotTESTreg.map(lambda x: x.label)
features = oneHotTESTreg.map(lambda x: x.features)
scaler = StandardScaler(withMean=False, withStd=True).fit(features)
dataTMP = label.zip(scaler.transform(features.map(lambda x: Vectors.dense(x.toArray()))))
oneHotTESTregScaled = dataTMP.map(lambda x: LabeledPoint(x[0], x[1]))

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 11.67 seconds

Cache objects in memory

The time taken for training and testing of ML algorithms can be reduced by caching the input data frame objects used for classification, regression and, scaled features.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# FOR BINARY CLASSIFICATION TRAINING AND TESTING
indexedTRAINbinary.cache()
indexedTESTbinary.cache()
oneHotTRAINbinary.cache()
oneHotTESTbinary.cache()

# FOR REGRESSION TRAINING AND TESTING
indexedTRAINreg.cache()
indexedTESTreg.cache()
oneHotTRAINreg.cache()
oneHotTESTreg.cache()

# SCALED FEATURES
oneHotTRAINregScaled.cache()
oneHotTESTregScaled.cache()

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 0.13 second

Predict whether or not a tip is paid with binary classification models

This section shows how use three models for the binary classification task of predicting whether or not a tip is paid for a taxi trip. The models presented are:

- Logistic regression
- Random forest
- Gradient Boosting Trees

Each model building code section is split into steps:

1. **Model training** data with one parameter set
2. **Model evaluation** on a test data set with metrics
3. **Saving model** in blob for future consumption

We show how to do cross-validation (CV) with parameter sweeping in two ways:

1. Using **generic** custom code that can be applied to any algorithm in MLlib and to any parameter sets in an algorithm.
2. Using the **pySpark CrossValidator pipeline function**. CrossValidator has a few limitations for Spark 1.5.0:
   - Pipeline models cannot be saved or persisted for future consumption.
   - Cannot be used for every parameter in a model.
   - Cannot be used for every MLlib algorithm.

Generic cross validation and hyperparameter sweeping used with the logistic regression algorithm for binary classification

The code in this section shows how to train, evaluate, and save a logistic regression model with LBFGS that predicts whether or not a tip is paid for a trip in the NYC taxi trip and fare dataset. The model is trained using cross validation (CV) and hyperparameter sweeping implemented with custom code that can be applied to any of the learning algorithms in MLlib.

> **NOTE**
>
> The execution of this custom CV code can take several minutes.

Train the logistic regression model using CV and hyperparameter sweeping

```
# LOGISTIC REGRESSION CLASSIFICATION WITH CV AND HYPERPARAMETER SWEEPING

# GET ACCURACY FOR HYPERPARAMETERS BASED ON CROSS-VALIDATION IN TRAINING DATA-SET

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD LIBRARIES
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.evaluation import BinaryClassificationMetrics

# CREATE PARAMETER GRID FOR LOGISTIC REGRESSION PARAMETER SWEEP
from sklearn.grid_search import ParameterGrid
grid = [{'regParam': [0.01, 0.1], 'iterations': [5, 10], 'regType': ["l1", "l2"], 'tolerance': [1e-3, 1e-
4]}]
paramGrid = list(ParameterGrid(grid))
numModels = len(paramGrid)

# SET NUM FOLDS AND NUM PARAMETER SETS TO SWEEP ON
```

```
nFolds = 3;
h = 1.0 / nFolds;
metricSum = np.zeros(numModels);

# BEGIN CV WITH PARAMETER SWEEP
for i in range(nFolds):
    # Create training and x-validation sets
    validateLB = i * h
    validateUB = (i + 1) * h
    condition = (trainData["rand"] >= validateLB) & (trainData["rand"] < validateUB)
    validation = trainData.filter(condition)
    # Create LabeledPoints from data-frames
    if i > 0:
        trainCVLabPt.unpersist()
        validationLabPt.unpersist()
    trainCV = trainData.filter(~condition)
    trainCVLabPt = trainCV.map(parseRowOneHotBinary)
    trainCVLabPt.cache()
    validationLabPt = validation.map(parseRowOneHotBinary)
    validationLabPt.cache()
    # For parameter sets compute metrics from x-validation
    for j in range(numModels):
        regt = paramGrid[j]['regType']
        regp = paramGrid[j]['regParam']
        iters = paramGrid[j]['iterations']
        tol = paramGrid[j]['tolerance']
        # Train logistic regression model with hypermarameter set
        model = LogisticRegressionWithLBFGS.train(trainCVLabPt, regType=regt, iterations=iters,
                                                  regParam=regp, tolerance = tol, intercept=True)
        predictionAndLabels = validationLabPt.map(lambda lp: (float(model.predict(lp.features)), lp.label))
        # Use ROC-AUC as accuracy metrics
        validMetrics = BinaryClassificationMetrics(predictionAndLabels)
        metric = validMetrics.areaUnderROC
        metricSum[j] += metric

avgAcc = metricSum / nFolds;
bestParam = paramGrid[np.argmax(avgAcc)];

# UNPERSIST OBJECTS
trainCVLabPt.unpersist()
validationLabPt.unpersist()

# TRAIN ON FULL TRAIING SET USING BEST PARAMETERS FROM CV/PARAMETER SWEEP
logitBest = LogisticRegressionWithLBFGS.train(oneHotTRAINbinary, regType=bestParam['regType'],
                                              iterations=bestParam['iterations'],
                                              regParam=bestParam['regParam'], tolerance =
bestParam['tolerance'],
                                              intercept=True)

# PRINT COEFFICIENTS AND INTERCEPT OF THE MODEL
# NOTE: There are 20 coefficient terms for the 10 features,
#       and the different categories for features: vendorVec (2), rateVec, paymentVec (6),
TrafficTimeBinsVec (4)
print("Coefficients: " + str(logitBest.weights))
print("Intercept: " + str(logitBest.intercept))

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Coefficients: [0.0082065285375, -0.0223675576104, -0.0183812028036, -3.48124578069e-05, -0.00247646947233, -0.00165897881503, 0.0675394837328, -0.111823113101, -0.324609912762, -0.204549780032, -1.36499216354, 0.591088507921, -0.664263411392, -1.00439726852, 3.46567827545, -3.51025855172, -0.0471341112232, -0.043521833294, 0.000243375810385, 0.054518719222]

Intercept: -0.0111216486893

Time taken to execute above cell: 14.43 seconds

Evaluate the binary classification model with standard metrics

The code in this section shows how to evaluate a logistic regression model against a test data-set, including a plot of the ROC curve.

```
# RECORD START TIME
timestart = datetime.datetime.now()

#IMPORT LIBRARIES
from sklearn.metrics import roc_curve,auc
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.mllib.evaluation import MulticlassMetrics

# PREDICT ON TEST DATA WITH BEST/FINAL MODEL
predictionAndLabels = oneHotTESTbinary.map(lambda lp: (float(logitBest.predict(lp.features)), lp.label))

# INSTANTIATE METRICS OBJECT
metrics = BinaryClassificationMetrics(predictionAndLabels)

# AREA UNDER PRECISION-RECALL CURVE
print("Area under PR = %s" % metrics.areaUnderPR)

# AREA UNDER ROC CURVE
print("Area under ROC = %s" % metrics.areaUnderROC)
metrics = MulticlassMetrics(predictionAndLabels)

# OVERALL STATISTICS
precision = metrics.precision()
recall = metrics.recall()
f1Score = metrics.fMeasure()
print("Summary Stats")
print("Precision = %s" % precision)
print("Recall = %s" % recall)
print("F1 Score = %s" % f1Score)

# OUTPUT PROBABILITIES AND REGISTER TEMP TABLE
logitBest.clearThreshold(); # This clears threshold for classification (0.5) and outputs probabilities
predictionAndLabelsDF = predictionAndLabels.toDF()
predictionAndLabelsDF.registerTempTable("tmp_results");

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Area under PR = 0.985336538462

Area under ROC = 0.983383274312

Summary Stats

Precision = 0.984174341679

Recall = 0.984174341679

F1 Score = 0.984174341679

Time taken to execute above cell: 2.67 seconds

Plot the ROC curve.

The *predictionAndLabelsDF* is registered as a table, *tmp_results*, in the previous cell. *tmp_results* can be used to do queries and output results into the sqlResults data-frame for plotting. Here is the code.

```
# QUERY RESULTS
%%sql -q -o sqlResults
SELECT * from tmp_results
```

Here is the code to make predictions and plot the ROC-curve.

```
# MAKE PREDICTIONS AND PLOT ROC-CURVE

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
%matplotlib inline
from sklearn.metrics import roc_curve,auc

#PREDICTIONS
predictions_pddf = sqlResults.rename(columns={'_1': 'probability', '_2': 'label'})
prob = predictions_pddf["probability"]
fpr, tpr, thresholds = roc_curve(predictions_pddf['label'], prob, pos_label=1);
roc_auc = auc(fpr, tpr)

# PLOT ROC CURVES
plt.figure(figsize=(5,5))
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc="lower right")
plt.show()
```

OUTPUT

Persist model in a blob for future consumption

The code in this section shows how to save the logistic regression model for consumption.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.classification import LogisticRegressionModel

# PERSIST MODEL
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
logisticregressionfilename = "LogisticRegressionWithLBFGS_" + datestamp;
dirfilename = modelDir + logisticregressionfilename;

logitBest.save(sc, dirfilename);

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 34.57 seconds

Use MLlib's CrossValidator pipeline function with logistic regression (Elastic regression) model

The code in this section shows how to train, evaluate, and save a logistic regression model with LBFGS that predicts whether or not a tip is paid for a trip in the NYC taxi trip and fare dataset. The model is trained using cross validation (CV) and hyperparameter sweeping implemented with the MLlib CrossValidator pipeline function for CV with parameter sweep.

> **NOTE**
> The execution of this MLlib CV code can take several minutes.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from sklearn.metrics import roc_curve,auc

# DEFINE ALGORITHM / MODEL
lr = LogisticRegression()

# DEFINE GRID PARAMETERS
paramGrid = ParamGridBuilder().addGrid(lr.regParam, (0.01, 0.1))\
                              .addGrid(lr.maxIter, (5, 10))\
                              .addGrid(lr.tol, (1e-4, 1e-5))\
                              .addGrid(lr.elasticNetParam, (0.25,0.75))\
                              .build()

# DEFINE CV WITH PARAMETER SWEEP
cv = CrossValidator(estimator= lr,
                    estimatorParamMaps=paramGrid,
                    evaluator=BinaryClassificationEvaluator(),
                    numFolds=3)

# CONVERT TO DATA-FRAME: THIS DOES NOT RUN ON RDDs
trainDataFrame = sqlContext.createDataFrame(oneHotTRAINbinary, ["features", "label"])

# TRAIN WITH CROSS-VALIDATION
cv_model = cv.fit(trainDataFrame)

## PREDICT AND EVALUATE ON TEST DATA-SET

# USE TEST DATASET FOR PREDICTION
testDataFrame = sqlContext.createDataFrame(oneHotTESTbinary, ["features", "label"])
test_predictions = cv_model.transform(testDataFrame)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 107.98 seconds

Plot the ROC curve.

The *predictionAndLabelsDF* is registered as a table, *tmp_results*, in the previous cell. *tmp_results* can be used to do queries and output results into the sqlResults data-frame for plotting. Here is the code.

```
# QUERY RESULTS
%%sql -q -o sqlResults
SELECT label, prediction, probability from tmp_results
```

Here is the code to plot the ROC curve.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
from sklearn.metrics import roc_curve,auc

# ROC CURVE
prob = [x["values"][1] for x in sqlResults["probability"]]
fpr, tpr, thresholds = roc_curve(sqlResults['label'], prob, pos_label=1);
roc_auc = auc(fpr, tpr)

#PLOT
plt.figure(figsize=(5,5))
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc="lower right")
plt.show()
```

OUTPUT

Random forest classification

The code in this section shows how to train, evaluate, and save a random forest regression that predicts whether or not a tip is paid for a trip in the NYC taxi trip and fare dataset.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.mllib.evaluation import MulticlassMetrics

# SPECIFY NUMBER OF CATEGORIES FOR CATEGORICAL FEATURES. FEATURE #0 HAS 2 CATEGORIES, FEATURE #2 HAS 2
CATEGORIES, AND SO ON
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}

# TRAIN RANDOMFOREST MODEL
rfModel = RandomForest.trainClassifier(indexedTRAINbinary, numClasses=2,
                                       categoricalFeaturesInfo=categoricalFeaturesInfo,
                                       numTrees=25, featureSubsetStrategy="auto",
                                       impurity='gini', maxDepth=5, maxBins=32)
## UN-COMMENT IF YOU WANT TO PRING TREES
#print('Learned classification forest model:')
#print(rfModel.toDebugString())

# PREDICT ON TEST DATA AND EVALUATE
predictions = rfModel.predict(indexedTESTbinary.map(lambda x: x.features))
predictionAndLabels = indexedTESTbinary.map(lambda lp: lp.label).zip(predictions)

# AREA UNDER ROC CURVE
metrics = BinaryClassificationMetrics(predictionAndLabels)
print("Area under ROC = %s" % metrics.areaUnderROC)

# PERSIST MODEL IN BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
rfclassificationfilename = "RandomForestClassification_" + datestamp;
dirfilename = modelDir + rfclassificationfilename;

rfModel.save(sc, dirfilename);

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Area under ROC = 0.985336538462

Time taken to execute above cell: 26.72 seconds

Gradient boosting trees classification

The code in this section shows how to train, evaluate, and save a gradient boosting trees model that predicts whether or not a tip is paid for a trip in the NYC taxi trip and fare dataset.

```
# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel

# SPECIFY NUMBER OF CATEGORIES FOR CATEGORICAL FEATURES. FEATURE #0 HAS 2 CATEGORIES, FEATURE #2 HAS 2
CATEGORIES, AND SO ON
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}

gbtModel = GradientBoostedTrees.trainClassifier(indexedTRAINbinary,
categoricalFeaturesInfo=categoricalFeaturesInfo,
                                                numIterations=10)
## UNCOMMENT IF YOU WANT TO PRINT TREE DETAILS
#print('Learned classification GBT model:')
#print(bgtModel.toDebugString())

# PREDICT ON TEST DATA AND EVALUATE
predictions = gbtModel.predict(indexedTESTbinary.map(lambda x: x.features))
predictionAndLabels = indexedTESTbinary.map(lambda lp: lp.label).zip(predictions)

# Area under ROC curve
metrics = BinaryClassificationMetrics(predictionAndLabels)
print("Area under ROC = %s" % metrics.areaUnderROC)

# PERSIST MODEL IN A BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
btclassificationfilename = "GradientBoostingTreeClassification_" + datestamp;
dirfilename = modelDir + btclassificationfilename;

gbtModel.save(sc, dirfilename)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Area under ROC = 0.985336538462

Time taken to execute above cell: 28.13 seconds

Predict tip amount with regression models (not using CV)

This section shows how use three models for the regression task: predict the tip amount paid for a taxi trip based on other tip features. The models presented are:

- Regularized linear regression
- Random forest
- Gradient Boosting Trees

These models were described in the introduction. Each model building code section is split into steps:

1. **Model training** data with one parameter set
2. **Model evaluation** on a test data set with metrics
3. **Saving model** in blob for future consumption

> **NOTE**
> Cross-validation is not used with the three regression models in this section, since this was shown in detail for the logistic regression models. An example showing how to use CV with Elastic Net for linear regression is provided in the Appendix of this topic.

> **NOTE**
> In our experience, there can be issues with convergence of LinearRegressionWithSGD models, and parameters need to be changed/optimized carefully for obtaining a valid model. Scaling of variables significantly helps with convergence. Elastic net regression, shown in the Appendix to this topic, can also be used instead of LinearRegressionWithSGD.

Linear regression with SGD

The code in this section shows how to use scaled features to train a linear regression that uses stochastic gradient descent (SGD) for optimization, and how to score, evaluate, and save the model in Azure Blob Storage (WASB).

> **TIP**
> In our experience, there can be issues with the convergence of LinearRegressionWithSGD models, and parameters need to be changed/optimized carefully for obtaining a valid model. Scaling of variables significantly helps with convergence.

```
# LINEAR REGRESSION WITH SGD

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD LIBRARIES
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD, LinearRegressionModel
from pyspark.mllib.evaluation import RegressionMetrics
from scipy import stats

# USE SCALED FEATURES TO TRAIN MODEL
linearModel = LinearRegressionWithSGD.train(oneHotTRAINregScaled, iterations=100, step = 0.1, regType='l2',
regParam=0.1, intercept = True)

# PRINT COEFFICIENTS AND INTERCEPT OF THE MODEL
# NOTE: There are 20 coefficient terms for the 10 features,
#       and the different categories for features: vendorVec (2), rateVec, paymentVec (6),
TrafficTimeBinsVec (4)
print("Coefficients: " + str(linearModel.weights))
print("Intercept: " + str(linearModel.intercept))

# SCORE ON SCALED TEST DATA-SET & EVALUATE
predictionAndLabels = oneHotTESTregScaled.map(lambda lp: (float(linearModel.predict(lp.features)),
lp.label))
testMetrics = RegressionMetrics(predictionAndLabels)

print("RMSE = %s" % testMetrics.rootMeanSquaredError)
print("R-sqr = %s" % testMetrics.r2)

# SAVE MODEL IN BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
linearregressionfilename = "LinearRegressionWithSGD_" + datestamp;
dirfilename = modelDir + linearregressionfilename;

linearModel.save(sc, dirfilename)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Coefficients: [0.0141707753435, -0.0252930927087, -0.0231442517137, 0.247070902996, 0.312544147152, 0.360296120645, 0.0122079566092, -0.00456498588241, -0.0898228505177, 0.0714046248793, 0.102171263868, 0.100022455632, -0.00289545676449, -0.00791124681938, 0.54396316518, -0.536293513569, 0.0119076553369, -0.0173039244582, 0.0119632796147, 0.00146764882502]

Intercept: 0.854507624459

RMSE = 1.23485131376

R-sqr = 0.597963951127

Time taken to execute above cell: 38.62 seconds

Random Forest regression

The code in this section shows how to train, evaluate, and save a random forest model that predicts tip amount for the NYC taxi trip data.

NOTE

Cross-validation with parameter sweeping using custom code is provided in the appendix.

```
#PREDICT TIP AMOUNTS USING RANDOM FOREST

# RECORD START TIME
timestart= datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils
from pyspark.mllib.evaluation import RegressionMetrics

# TRAIN MODEL
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}
rfModel = RandomForest.trainRegressor(indexedTRAINreg, categoricalFeaturesInfo=categoricalFeaturesInfo,
                                      numTrees=25, featureSubsetStrategy="auto",
                                      impurity='variance', maxDepth=10, maxBins=32)
# UN-COMMENT IF YOU WANT TO PRING TREES
#print('Learned classification forest model:')
#print(rfModel.toDebugString())

# PREDICT AND EVALUATE ON TEST DATA-SET
predictions = rfModel.predict(indexedTESTreg.map(lambda x: x.features))
predictionAndLabels = oneHotTESTreg.map(lambda lp: lp.label).zip(predictions)

testMetrics = RegressionMetrics(predictionAndLabels)
print("RMSE = %s" % testMetrics.rootMeanSquaredError)
print("R-sqr = %s" % testMetrics.r2)

# SAVE MODEL IN BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
rfregressionfilename = "RandomForestRegression_" + datestamp;
dirfilename = modelDir + rfregressionfilename;

rfModel.save(sc, dirfilename);

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

RMSE = 0.931981967875

R-sqr = 0.733445485802

Time taken to execute above cell: 25.98 seconds

Gradient boosting trees regression

The code in this section shows how to train, evaluate, and save a gradient boosting trees model that predicts tip amount for the NYC taxi trip data.

Train and evaluate

```
#PREDICT TIP AMOUNTS USING GRADIENT BOOSTING TREES

# RECORD START TIME
timestart= datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel
from pyspark.mllib.util import MLUtils

# TRAIN MODEL
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}
gbtModel = GradientBoostedTrees.trainRegressor(indexedTRAINreg,
categoricalFeaturesInfo=categoricalFeaturesInfo,
                                                numIterations=10, maxBins=32, maxDepth = 4,
learningRate=0.1)

# EVALUATE A TEST DATA-SET
predictions = gbtModel.predict(indexedTESTreg.map(lambda x: x.features))
predictionAndLabels = indexedTESTreg.map(lambda lp: lp.label).zip(predictions)

testMetrics = RegressionMetrics(predictionAndLabels)
print("RMSE = %s" % testMetrics.rootMeanSquaredError)
print("R-sqr = %s" % testMetrics.r2)

# PLOT SCATTER-PLOT BETWEEN ACTUAL AND PREDICTED TIP VALUES
test_predictions= sqlContext.createDataFrame(predictionAndLabels)
test_predictions_pddf = test_predictions.toPandas()

# SAVE MODEL IN BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
btregressionfilename = "GradientBoostingTreeRegression_" + datestamp;
dirfilename = modelDir + btregressionfilename;
gbtModel.save(sc, dirfilename)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

RMSE = 0.928172197114

R-sqr = 0.732680354389

Time taken to execute above cell: 20.9 seconds

Plot

tmp_results is registered as a Hive table in the previous cell. Results from the table are output into the *sqlResults* data-frame for plotting. Here is the code

```
# PLOT SCATTER-PLOT BETWEEN ACTUAL AND PREDICTED TIP VALUES

# SELECT RESULTS
%%sql -q -o sqlResults
SELECT * from tmp_results
```

Here is the code to plot the data using the Jupyter server.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
import numpy as np

# PLOT
ax = sqlResults.plot(kind='scatter', figsize = (6,6), x='_1', y='_2', color='blue', alpha = 0.25,
label='Actual vs. predicted');
fit = np.polyfit(sqlResults['_1'], sqlResults['_2'], deg=1)
ax.set_title('Actual vs. Predicted Tip Amounts ($)')
ax.set_xlabel("Actual")
ax.set_ylabel("Predicted")
ax.plot(sqlResults['_1'], fit[0] * sqlResults['_1'] + fit[1], color='magenta')
plt.axis([-1, 15, -1, 15])
plt.show(ax)
```

Appendix: Additional regression tasks using cross validation with parameter sweeps

This appendix contains code showing how to do CV using Elastic net for linear regression and how to do CV with parameter sweep using custom code for random forest regression.

Cross validation using Elastic net for linear regression

The code in this section shows how to do cross validation using Elastic net for linear regression and how to evaluate the model against test data.

```
###  CV USING ELASTIC NET FOR LINEAR REGRESSION

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# DEFINE ALGORITHM/MODEL
lr = LinearRegression()

# DEFINE GRID PARAMETERS
paramGrid = ParamGridBuilder().addGrid(lr.regParam, (0.01, 0.1))\
                              .addGrid(lr.maxIter, (5, 10))\
                              .addGrid(lr.tol, (1e-4, 1e-5))\
                              .addGrid(lr.elasticNetParam, (0.25,0.75))\
                              .build()

# DEFINE PIPELINE
# SIMPLY THE MODEL HERE, WITHOUT TRANSFORMATIONS
pipeline = Pipeline(stages=[lr])

# DEFINE CV WITH PARAMETER SWEEP
cv = CrossValidator(estimator= lr,
                    estimatorParamMaps=paramGrid,
                    evaluator=RegressionEvaluator(),
                    numFolds=3)

# CONVERT TO DATA FRAME, AS CROSSVALIDATOR WON'T RUN ON RDDS
trainDataFrame = sqlContext.createDataFrame(oneHotTRAINreg, ["features", "label"])

# TRAIN WITH CROSS-VALIDATION
cv_model = cv.fit(trainDataFrame)

# EVALUATE MODEL ON TEST SET
testDataFrame = sqlContext.createDataFrame(oneHotTESTreg, ["features", "label"])

# MAKE PREDICTIONS ON TEST DOCUMENTS
# cvModel uses the best model found (lrModel).
predictionAndLabels = cv_model.transform(testDataFrame)

# CONVERT TO DF AND SAVE REGISTER DF AS TABLE
predictionAndLabels.registerTempTable("tmp_results");

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

Time taken to execute above cell: 161.21 seconds

Evaluate with R-SQR metric

tmp_results is registered as a Hive table in the previous cell. Results from the table are output into the *sqlResults* data-frame for plotting. Here is the code

```
# SELECT RESULTS
%%sql -q -o sqlResults
SELECT label,prediction from tmp_results
```

Here is the code to calculate R-sqr.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
from scipy import stats

#R-SQR TEST METRIC
corstats = stats.linregress(sqlResults['label'],sqlResults['prediction'])
r2 = (corstats[2]*corstats[2])
print("R-sqr = %s" % r2)
```

OUTPUT

R-sqr = 0.619184907088

Cross validation with parameter sweep using custom code for random forest regression

The code in this section shows how to do cross validation with parameter sweep using custom code for random forest regression and how to evaluate the model against test data.

```
# RECORD START TIME
timestart= datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
# GET ACCURARY FOR HYPERPARAMETERS BASED ON CROSS-VALIDATION IN TRAINING DATA-SET
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils
from pyspark.mllib.evaluation import RegressionMetrics
from sklearn.grid_search import ParameterGrid

## CREATE PARAMETER GRID
grid = [{'maxDepth': [5,10], 'numTrees': [25,50]}]
paramGrid = list(ParameterGrid(grid))

## SPECIFY LEVELS OF CATEGORICAL VARIBLES
categoricalFeaturesInfo={0:2, 1:2, 2:6, 3:4}

# SPECIFY NUMFOLDS AND ARRAY TO HOLD METRICS
nFolds = 3;
numModels = len(paramGrid)
h = 1.0 / nFolds;
metricSum = np.zeros(numModels);

for i in range(nFolds):
    # Create training and x-validation sets
    validateLB = i * h
    validateUB = (i + 1) * h
    condition = (trainData["rand"] >= validateLB) & (trainData["rand"] < validateUB)
    validation = trainData.filter(condition)
    # Create labeled points from data-frames
    if i > 0:
        trainCVLabPt.unpersist()
        validationLabPt.unpersist()
    trainCV = trainData.filter(~condition)
    trainCVLabPt = trainCV.map(parseRowIndexingRegression)
    trainCVLabPt.cache()
    validationLabPt = validation.map(parseRowIndexingRegression)
    validationLabPt.cache()
    # For parameter sets compute metrics from x-validation
    for j in range(numModels):
        maxD = paramGrid[j]['maxDepth']
        numT = paramGrid[j]['numTrees']
        # Train logistic regression model with hypermarameter set
        rfModel = RandomForest.trainRegressor(trainCVLabPt, categoricalFeaturesInfo=categoricalFeaturesInfo,
                                              numTrees=numT, featureSubsetStrategy="auto",
                                              impurity='variance', maxDepth=maxD, maxBins=32)
```

```
        predictions = rfModel.predict(validationLabPt.map(lambda x: x.features))
        predictionAndLabels = validationLabPt.map(lambda lp: lp.label).zip(predictions)
        # Use ROC-AUC as accuracy metrics
        validMetrics = RegressionMetrics(predictionAndLabels)
        metric = validMetrics.rootMeanSquaredError
        metricSum[j] += metric

avgAcc = metricSum/nFolds;
bestParam = paramGrid[np.argmin(avgAcc)];

# UNPERSIST OBJECTS
trainCVLabPt.unpersist()
validationLabPt.unpersist()

## TRAIN FINAL MODL WITH BEST PARAMETERS
rfModel = RandomForest.trainRegressor(indexedTRAINreg, categoricalFeaturesInfo=categoricalFeaturesInfo,
                                      numTrees=bestParam['numTrees'], featureSubsetStrategy="auto",
                                      impurity='variance', maxDepth=bestParam['maxDepth'], maxBins=32)

# EVALUATE MODEL ON TEST DATA
predictions = rfModel.predict(indexedTESTreg.map(lambda x: x.features))
predictionAndLabels = indexedTESTreg.map(lambda lp: lp.label).zip(predictions)

#PRINT TEST METRICS
testMetrics = RegressionMetrics(predictionAndLabels)
print("RMSE = %s" % testMetrics.rootMeanSquaredError)
print("R-sqr = %s" % testMetrics.r2)

# PRINT ELAPSED TIME
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT

RMSE = 0.906972198262

R-sqr = 0.740751197012

Time taken to execute above cell: 69.17 seconds

Clean up objects from memory and print model locations

Use `unpersist()` to delete objects cached in memory.

```
# UNPERSIST OBJECTS CACHED IN MEMORY

# REMOVE ORIGINAL DFs
taxi_df_train_cleaned.unpersist()
taxi_df_train_with_newFeatures.unpersist()
trainData.unpersist()
trainData.unpersist()

# FOR BINARY CLASSIFICATION TRAINING AND TESTING
indexedTRAINbinary.unpersist()
indexedTESTbinary.unpersist()
oneHotTRAINbinary.unpersist()
oneHotTESTbinary.unpersist()

# FOR REGRESSION TRAINING AND TESTING
indexedTRAINreg.unpersist()
indexedTESTreg.unpersist()
oneHotTRAINreg.unpersist()
oneHotTESTreg.unpersist()

# SCALED FEATURES
oneHotTRAINregScaled.unpersist()
oneHotTESTregScaled.unpersist()
```

OUTPUT

PythonRDD[122] at RDD at PythonRDD.scala: 43

Output path to model files to be used in the consumption notebook. To consume and score an independent data-set, you need to copy and paste these file names in the "Consumption notebook".

```
# PRINT MODEL FILE LOCATIONS FOR CONSUMPTION
print "logisticRegFileLoc = modelDir + \"" + logisticregressionfilename + "\"";
print "linearRegFileLoc = modelDir + \"" + linearregressionfilename + "\"";
print "randomForestClassificationFileLoc = modelDir + \"" + rfclassificationfilename + "\"";
print "randomForestRegFileLoc = modelDir + \"" + rfregressionfilename + "\"";
print "BoostedTreeClassificationFileLoc = modelDir + \"" + btclassificationfilename + "\"";
print "BoostedTreeRegressionFileLoc = modelDir + \"" + btregressionfilename + "\"";
```

OUTPUT

logisticRegFileLoc = modelDir + "LogisticRegressionWithLBFGS_2016-05-0316_47_30.096528"

linearRegFileLoc = modelDir + "LinearRegressionWithSGD_2016-05-0316_51_28.433670"

randomForestClassificationFileLoc = modelDir + "RandomForestClassification_2016-05-0316_50_17.454440"

randomForestRegFileLoc = modelDir + "RandomForestRegression_2016-05-0316_51_57.331730"

BoostedTreeClassificationFileLoc = modelDir + "GradientBoostingTreeClassification_2016-05-0316_50_40.138809"

BoostedTreeRegressionFileLoc = modelDir + "GradientBoostingTreeRegression_2016-05-0316_52_18.827237"

What's next?

Now that you have created regression and classification models with the Spark MlLib, you are ready to learn how to score and evaluate these models.

Model consumption: To learn how to score and evaluate the classification and regression models created in this topic, see Score and evaluate Spark-built machine learning models.

Operationalize Spark-built machine learning models

11/19/2021 • 17 minutes to read • Edit Online

This topic shows how to operationalize a saved machine learning model (ML) using Python on HDInsight Spark clusters. It describes how to load machine learning models that have been built using Spark MLlib and stored in Azure Blob Storage (WASB), and how to score them with datasets that have also been stored in WASB. It shows how to pre-process the input data, transform features using the indexing and encoding functions in the MLlib toolkit, and how to create a labeled point data object that can be used as input for scoring with the ML models. The models used for scoring include Linear Regression, Logistic Regression, Random Forest Models, and Gradient Boosting Tree Models.

Spark clusters and Jupyter notebooks

Setup steps and the code to operationalize an ML model are provided in this walkthrough for using an HDInsight Spark 1.6 cluster as well as a Spark 2.0 cluster. The code for these procedures is also provided in Jupyter notebooks.

Notebook for Spark 1.6

The pySpark-machine-learning-data-science-spark-model-consumption.ipynb Jupyter notebook shows how to operationalize a saved model using Python on HDInsight clusters.

Notebook for Spark 2.0

To modify the Jupyter notebook for Spark 1.6 to use with an HDInsight Spark 2.0 cluster, replace the Python code file with this file. This code shows how to consume the models created in Spark 2.0.

Prerequisites

1. You need an Azure account and a Spark 1.6 (or Spark 2.0) HDInsight cluster to complete this walkthrough. See the Overview of Data Science using Spark on Azure HDInsight for instructions on how to satisfy these requirements. That topic also contains a description of the NYC 2013 Taxi data used here and instructions on how to execute code from a Jupyter notebook on the Spark cluster.
2. Create the machine learning models to be scored here by working through the Data exploration and modeling with Spark topic for the Spark 1.6 cluster or the Spark 2.0 notebooks.
3. The Spark 2.0 notebooks use an additional data set for the classification task, the well-known Airline On-time departure dataset from 2011 and 2012. A description of the notebooks and links to them are provided in the Readme.md for the GitHub repository containing them. Moreover, the code here and in the linked notebooks is generic and should work on any Spark cluster. If you are not using HDInsight Spark, the cluster setup and management steps may be slightly different from what is shown here.

> **WARNING**
>
> Billing for HDInsight clusters is prorated per minute, whether you use them or not. Be sure to delete your cluster after you finish using it. See how to delete an HDInsight cluster.

Setup: storage locations, libraries, and the preset Spark context

Spark is able to read and write to an Azure Storage Blob (WASB). So any of your existing data stored there can be processed using Spark and the results stored again in WASB.

To save models or files in WASB, the path needs to be specified properly. The default container attached to the Spark cluster can be referenced using a path beginning with: *"wasb:///"*. The following code sample specifies the location of the data to be read and the path for the model storage directory to which the model output is saved.

Set directory paths for storage locations in WASB

Models are saved in: "wasb:///user/remoteuser/NYCTaxi/Models". If this path is not set properly, models are not loaded for scoring.

The scored results have been saved in: "wasb:///user/remoteuser/NYCTaxi/ScoredResults". If the path to folder is incorrect, results are not saved in that folder.

> **NOTE**
>
> The file path locations can be copied and pasted into the placeholders in this code from the output of the last cell of the **machine-learning-data-science-spark-data-exploration-modeling.ipynb** notebook.

Here is the code to set directory paths:

```
# LOCATION OF DATA TO BE SCORED (TEST DATA)
taxi_test_file_loc =
"wasb://mllibwalkthroughs@cdspsparksamples.blob.core.windows.net/Data/NYCTaxi/JoinedTaxiTripFare.Point1Pct.T
est.tsv";

# SET THE MODEL STORAGE DIRECTORY PATH
# NOTE THE LAST BACKSLASH IN THIS PATH IS NEEDED
modelDir = "wasb:///user/remoteuser/NYCTaxi/Models/"

# SET SCORDED RESULT DIRECTORY PATH
# NOTE THE LAST BACKSLASH IN THIS PATH IS NEEDED
scoredResultDir = "wasb:///user/remoteuser/NYCTaxi/ScoredResults/";

# FILE LOCATIONS FOR THE MODELS TO BE SCORED
logisticRegFileLoc = modelDir + "LogisticRegressionWithLBFGS_2016-04-1817_40_35.796789"
linearRegFileLoc = modelDir + "LinearRegressionWithSGD_2016-04-1817_44_00.993832"
randomForestClassificationFileLoc = modelDir + "RandomForestClassification_2016-04-1817_42_58.899412"
randomForestRegFileLoc = modelDir + "RandomForestRegression_2016-04-1817_44_27.204734"
BoostedTreeClassificationFileLoc = modelDir + "GradientBoostingTreeClassification_2016-04-1817_43_16.354770"
BoostedTreeRegressionFileLoc = modelDir + "GradientBoostingTreeRegression_2016-04-1817_44_46.206262"

# RECORD START TIME
import datetime
datetime.datetime.now()
```

OUTPUT:

datetime.datetime(2016, 4, 25, 23, 56, 19, 229403)

Import libraries

Set spark context and import necessary libraries with the following code

```
#IMPORT LIBRARIES
import pyspark
from pyspark import SparkConf
from pyspark import SparkContext
from pyspark.sql import SQLContext
import matplotlib
import matplotlib.pyplot as plt
from pyspark.sql import Row
from pyspark.sql.functions import UserDefinedFunction
from pyspark.sql.types import *
import atexit
from numpy import array
import numpy as np
import datetime
```

Preset Spark context and PySpark magics

The PySpark kernels that are provided with Jupyter notebooks have a preset context. Therefore, you do not need to set the Spark or Hive contexts explicitly before you start working with the application you are developing. These contexts are available by default:

- sc - for Spark
- sqlContext - for Hive

The PySpark kernel provides some predefined "magics", which are special commands that you can call with %%. There are two such commands that are used in these code samples.

- **%%local** Specified that the code in subsequent lines is executed locally. Code must be valid Python code.
- **%%sql -o <variable name>**
- Executes a Hive query against the sqlContext. If the -o parameter is passed, the result of the query is persisted in the %%local Python context as a Pandas dataframe.

For more information on the kernels for Jupyter notebooks and the predefined "magics" that they provide, see Kernels available for Jupyter notebooks with HDInsight Spark Linux clusters on HDInsight.

Ingest data and create a cleaned data frame

This section contains the code for a series of tasks required to ingest the data to be scored. Read in a joined 0.1% sample of the taxi trip and fare file (stored as a .tsv file), format the data, and then creates a clean data frame.

The taxi trip and fare files were joined based on the procedure provided in the: The Team Data Science Process in action: using HDInsight Hadoop clusters topic.

```
# INGEST DATA AND CREATE A CLEANED DATA FRAME

# RECORD START TIME
timestart = datetime.datetime.now()

# IMPORT FILE FROM PUBLIC BLOB
taxi_test_file = sc.textFile(taxi_test_file_loc)

# GET SCHEMA OF THE FILE FROM HEADER
taxi_header = taxi_test_file.filter(lambda l: "medallion" in l)

# PARSE FIELDS AND CONVERT DATA TYPE FOR SOME FIELDS
taxi_temp = taxi_test_file.subtract(taxi_header).map(lambda k: k.split("\t"))\
        .map(lambda p: (p[0],p[1],p[2],p[3],p[4],p[5],p[6],int(p[7]),int(p[8]),int(p[9]),int(p[10]),
                        float(p[11]),float(p[12]),p[13],p[14],p[15],p[16],p[17],p[18],float(p[19]),

float(p[20]),float(p[21]),float(p[22]),float(p[23]),float(p[24]),int(p[25]),int(p[26])))

# GET SCHEMA OF THE FILE FROM HEADER
schema_string = taxi_test_file.first()
fields = [StructField(field_name, StringType(), True) for field_name in schema_string.split('\t')]
fields[7].dataType = IntegerType() #Pickup hour
fields[8].dataType = IntegerType() # Pickup week
fields[9].dataType = IntegerType() # Weekday
fields[10].dataType = IntegerType() # Passenger count
fields[11].dataType = FloatType() # Trip time in secs
fields[12].dataType = FloatType() # Trip distance
fields[19].dataType = FloatType() # Fare amount
fields[20].dataType = FloatType() # Surcharge
fields[21].dataType = FloatType() # Mta_tax
fields[22].dataType = FloatType() # Tip amount
fields[23].dataType = FloatType() # Tolls amount
fields[24].dataType = FloatType() # Total amount
fields[25].dataType = IntegerType() # Tipped or not
fields[26].dataType = IntegerType() # Tip class
taxi_schema = StructType(fields)

# CREATE DATA FRAME
taxi_df_test = sqlContext.createDataFrame(taxi_temp, taxi_schema)

# CREATE A CLEANED DATA-FRAME BY DROPPING SOME UN-NECESSARY COLUMNS & FILTERING FOR UNDESIRED VALUES OR
OUTLIERS
taxi_df_test_cleaned =
taxi_df_test.drop('medallion').drop('hack_license').drop('store_and_fwd_flag').drop('pickup_datetime')\
    .drop('dropoff_datetime').drop('pickup_longitude').drop('pickup_latitude').drop('dropoff_latitude')\
    .drop('dropoff_longitude').drop('tip_class').drop('total_amount').drop('tolls_amount').drop('mta_tax')\
    .drop('direct_distance').drop('surcharge')\
    .filter("passenger_count > 0 and passenger_count < 8 AND payment_type in ('CSH', 'CRD') AND tip_amount
>= 0 AND tip_amount < 30 AND fare_amount >= 1 AND fare_amount < 150 AND trip_distance > 0 AND trip_distance
< 100 AND trip_time_in_secs > 30 AND trip_time_in_secs < 7200" )

# CACHE DATA-FRAME IN MEMORY & MATERIALIZE DF IN MEMORY
taxi_df_test_cleaned.cache()
taxi_df_test_cleaned.count()

# REGISTER DATA-FRAME AS A TEMP-TABLE IN SQL-CONTEXT
taxi_df_test_cleaned.registerTempTable("taxi_test")

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 46.37 seconds

Prepare data for scoring in Spark

This section shows how to index, encode, and scale categorical features to prepare them for use in MLlib supervised learning algorithms for classification and regression.

Feature transformation: index and encode categorical features for input into models for scoring

This section shows how to index categorical data using a `StringIndexer` and encode features with `OneHotEncoder` input into the models.

The StringIndexer encodes a string column of labels to a column of label indices. The indices are ordered by label frequencies.

The OneHotEncoder maps a column of label indices to a column of binary vectors, with at most a single one-value. This encoding allows algorithms that expect continuous valued features, such as logistic regression, to be applied to categorical features.

```
#INDEX AND ONE-HOT ENCODE CATEGORICAL FEATURES

# RECORD START TIME
timestart = datetime.datetime.now()

# LOAD PYSPARK LIBRARIES
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler, VectorIndexer

# CREATE FOUR BUCKETS FOR TRAFFIC TIMES
sqlStatement = """
    SELECT *,
    CASE
     WHEN (pickup_hour <= 6 OR pickup_hour >= 20) THEN "Night"
     WHEN (pickup_hour >= 7 AND pickup_hour <= 10) THEN "AMRush"
     WHEN (pickup_hour >= 11 AND pickup_hour <= 15) THEN "Afternoon"
     WHEN (pickup_hour >= 16 AND pickup_hour <= 19) THEN "PMRush"
    END as TrafficTimeBins
    FROM taxi_test
"""
taxi_df_test_with_newFeatures = sqlContext.sql(sqlStatement)

# CACHE DATA-FRAME IN MEMORY & MATERIALIZE DF IN MEMORY
taxi_df_test_with_newFeatures.cache()
taxi_df_test_with_newFeatures.count()

# INDEX AND ONE-HOT ENCODING
stringIndexer = StringIndexer(inputCol="vendor_id", outputCol="vendorIndex")
model = stringIndexer.fit(taxi_df_test_with_newFeatures) # Input data-frame is the cleaned one from above
indexed = model.transform(taxi_df_test_with_newFeatures)
encoder = OneHotEncoder(dropLast=False, inputCol="vendorIndex", outputCol="vendorVec")
encoded1 = encoder.transform(indexed)

# INDEX AND ENCODE RATE_CODE
stringIndexer = StringIndexer(inputCol="rate_code", outputCol="rateIndex")
model = stringIndexer.fit(encoded1)
indexed = model.transform(encoded1)
encoder = OneHotEncoder(dropLast=False, inputCol="rateIndex", outputCol="rateVec")
encoded2 = encoder.transform(indexed)

# INDEX AND ENCODE PAYMENT_TYPE
stringIndexer = StringIndexer(inputCol="payment_type", outputCol="paymentIndex")
model = stringIndexer.fit(encoded2)
indexed = model.transform(encoded2)
encoder = OneHotEncoder(dropLast=False, inputCol="paymentIndex", outputCol="paymentVec")
encoded3 = encoder.transform(indexed)

# INDEX AND ENCODE TRAFFIC TIME BINS
stringIndexer = StringIndexer(inputCol="TrafficTimeBins", outputCol="TrafficTimeBinsIndex")
model = stringIndexer.fit(encoded3)
indexed = model.transform(encoded3)
encoder = OneHotEncoder(dropLast=False, inputCol="TrafficTimeBinsIndex", outputCol="TrafficTimeBinsVec")
encodedFinal = encoder.transform(indexed)

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 5.37 seconds

Create RDD objects with feature arrays for input into models

This section contains code that shows how to index categorical text data as an RDD object and one-hot encode it so it can be used to train and test MLlib logistic regression and tree-based models. The indexed data is stored in

Resilient Distributed Dataset (RDD) objects. The RDDs are the basic abstraction in Spark. An RDD object represents an immutable, partitioned collection of elements that can be operated on in parallel with Spark.

It also contains code that shows how to scale data with the `StandardScalar` provided by MLlib for use in linear regression with Stochastic Gradient Descent (SGD), a popular algorithm for training a wide range of machine learning models. The StandardScaler is used to scale the features to unit variance. Feature scaling, also known as data normalization, insures that features with widely disbursed values are not given excessive weigh in the objective function.

```
# CREATE RDD OBJECTS WITH FEATURE ARRAYS FOR INPUT INTO MODELS

# RECORD START TIME
timestart = datetime.datetime.now()

# IMPORT LIBRARIES
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.feature import StandardScaler, StandardScalerModel
from pyspark.mllib.util import MLUtils
from numpy import array

# INDEXING CATEGORICAL TEXT FEATURES FOR INPUT INTO TREE-BASED MODELS
def parseRowIndexingBinary(line):
    features = np.array([line.paymentIndex, line.vendorIndex, line.rateIndex, line.TrafficTimeBinsIndex,
                         line.pickup_hour, line.weekday, line.passenger_count, line.trip_time_in_secs,
                         line.trip_distance, line.fare_amount])
    return  features

# ONE-HOT ENCODING OF CATEGORICAL TEXT FEATURES FOR INPUT INTO LOGISTIC REGRESSION MODELS
def parseRowOneHotBinary(line):
    features = np.concatenate((np.array([line.pickup_hour, line.weekday, line.passenger_count,
                                         line.trip_time_in_secs, line.trip_distance, line.fare_amount]),
                                         line.vendorVec.toArray(), line.rateVec.toArray(),
                                         line.paymentVec.toArray(), line.TrafficTimeBinsVec.toArray()),
axis=0)
    return  features

# ONE-HOT ENCODING OF CATEGORICAL TEXT FEATURES FOR INPUT INTO TREE-BASED MODELS
def parseRowIndexingRegression(line):
    features = np.array([line.paymentIndex, line.vendorIndex, line.rateIndex, line.TrafficTimeBinsIndex,
                         line.pickup_hour, line.weekday, line.passenger_count, line.trip_time_in_secs,
                         line.trip_distance, line.fare_amount])
    return  features

# INDEXING CATEGORICAL TEXT FEATURES FOR INPUT INTO LINEAR REGRESSION MODELS
def parseRowOneHotRegression(line):
    features = np.concatenate((np.array([line.pickup_hour, line.weekday, line.passenger_count,
                                         line.trip_time_in_secs, line.trip_distance, line.fare_amount]),
                                         line.vendorVec.toArray(), line.rateVec.toArray(),
                                         line.paymentVec.toArray(), line.TrafficTimeBinsVec.toArray()),
axis=0)
    return  features

# FOR BINARY CLASSIFICATION TRAINING AND TESTING
indexedTESTbinary = encodedFinal.map(parseRowIndexingBinary)
oneHotTESTbinary = encodedFinal.map(parseRowOneHotBinary)

# FOR REGRESSION CLASSIFICATION TRAINING AND TESTING
indexedTESTreg = encodedFinal.map(parseRowIndexingRegression)
oneHotTESTreg = encodedFinal.map(parseRowOneHotRegression)

# SCALING FEATURES FOR LINEARREGRESSIONWITHSGD MODEL
scaler = StandardScaler(withMean=False, withStd=True).fit(oneHotTESTreg)
oneHotTESTregScaled = scaler.transform(oneHotTESTreg)

# CACHE RDDS IN MEMORY
indexedTESTbinary.cache();
oneHotTESTbinary.cache();
indexedTESTreg.cache();
oneHotTESTreg.cache();
oneHotTESTregScaled.cache();

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 11.72 seconds

Score with the Logistic Regression Model and save output to blob

The code in this section shows how to load a Logistic Regression Model that has been saved in Azure blob storage and use it to predict whether or not a tip is paid on a taxi trip, score it with standard classification metrics, and then save and plot the results to blob storage. The scored results are stored in RDD objects.

```
# SCORE AND EVALUATE LOGISTIC REGRESSION MODEL

# RECORD START TIME
timestart = datetime.datetime.now()

# IMPORT LIBRARIES
from pyspark.mllib.classification import LogisticRegressionModel

## LOAD SAVED MODEL
savedModel = LogisticRegressionModel.load(sc, logisticRegFileLoc)
predictions = oneHotTESTbinary.map(lambda features: (float(savedModel.predict(features))))

## SAVE SCORED RESULTS (RDD) TO BLOB
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
logisticregressionfilename = "LogisticRegressionWithLBFGS_" + datestamp + ".txt";
dirfilename = scoredResultDir + logisticregressionfilename;
predictions.saveAsTextFile(dirfilename)

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 19.22 seconds

Score a Linear Regression Model

We used LinearRegressionWithSGD to train a linear regression model using Stochastic Gradient Descent (SGD) for optimization to predict the amount of tip paid.

The code in this section shows how to load a Linear Regression Model from Azure blob storage, score using scaled variables, and then save the results back to the blob.

```
#SCORE LINEAR REGRESSION MODEL

# RECORD START TIME
timestart = datetime.datetime.now()

#LOAD LIBRARIES
from pyspark.mllib.regression import LinearRegressionWithSGD, LinearRegressionModel

# LOAD MODEL AND SCORE USING ** SCALED VARIABLES **
savedModel = LinearRegressionModel.load(sc, linearRegFileLoc)
predictions = oneHotTESTregScaled.map(lambda features: (float(savedModel.predict(features))))

# SAVE RESULTS
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
linearregressionfilename = "LinearRegressionWithSGD_" + datestamp;
dirfilename = scoredResultDir + linearregressionfilename;
predictions.saveAsTextFile(dirfilename)

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 16.63 seconds

Score classification and regression Random Forest Models

The code in this section shows how to load the saved classification and regression Random Forest Models saved in Azure blob storage, score their performance with standard classifier and regression measures, and then save the results back to blob storage.

Random forests are ensembles of decision trees. They combine many decision trees to reduce the risk of overfitting. Random forests can handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions. Random forests are one of the most successful machine learning models for classification and regression.

spark.mllib supports random forests for binary and multiclass classification and for regression, using both continuous and categorical features.

```
# SCORE RANDOM FOREST MODELS FOR CLASSIFICATION AND REGRESSION

# RECORD START TIME
timestart = datetime.datetime.now()

#IMPORT MLLIB LIBRARIES
from pyspark.mllib.tree import RandomForest, RandomForestModel

# CLASSIFICATION: LOAD SAVED MODEL, SCORE AND SAVE RESULTS BACK TO BLOB
savedModel = RandomForestModel.load(sc, randomForestClassificationFileLoc)
predictions = savedModel.predict(indexedTESTbinary)

# SAVE RESULTS
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
rfclassificationfilename = "RandomForestClassification_" + datestamp + ".txt";
dirfilename = scoredResultDir + rfclassificationfilename;
predictions.saveAsTextFile(dirfilename)

# REGRESSION: LOAD SAVED MODEL, SCORE AND SAVE RESULTS BACK TO BLOB
savedModel = RandomForestModel.load(sc, randomForestRegFileLoc)
predictions = savedModel.predict(indexedTESTreg)

# SAVE RESULTS
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
rfregressionfilename = "RandomForestRegression_" + datestamp + ".txt";
dirfilename = scoredResultDir + rfregressionfilename;
predictions.saveAsTextFile(dirfilename)

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 31.07 seconds

Score classification and regression Gradient Boosting Tree Models

The code in this section shows how to load classification and regression Gradient Boosting Tree Models from Azure blob storage, score their performance with standard classifier and regression measures, and then save the results back to blob storage.

spark.mllib supports GBTS for binary classification and for regression, using both continuous and categorical features.

Gradient Boosting Trees (GBTS) are ensembles of decision trees. GBTS train decision trees iteratively to minimize a loss function. GBTS can handle categorical features, do not require feature scaling, and are able to capture non-linearities and feature interactions. This algorithm can also be used in a multiclass-classification setting.

```
# SCORE GRADIENT BOOSTING TREE MODELS FOR CLASSIFICATION AND REGRESSION

# RECORD START TIME
timestart = datetime.datetime.now()

#IMPORT MLLIB LIBRARIES
from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel

# CLASSIFICATION: LOAD SAVED MODEL, SCORE AND SAVE RESULTS BACK TO BLOB

#LOAD AND SCORE THE MODEL
savedModel = GradientBoostedTreesModel.load(sc, BoostedTreeClassificationFileLoc)
predictions = savedModel.predict(indexedTESTbinary)

# SAVE RESULTS
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
btclassificationfilename = "GradientBoostingTreeClassification_" + datestamp + ".txt";
dirfilename = scoredResultDir + btclassificationfilename;
predictions.saveAsTextFile(dirfilename)

# REGRESSION: LOAD SAVED MODEL, SCORE AND SAVE RESULTS BACK TO BLOB

# LOAD AND SCORE MODEL
savedModel = GradientBoostedTreesModel.load(sc, BoostedTreeRegressionFileLoc)
predictions = savedModel.predict(indexedTESTreg)

# SAVE RESULTS
datestamp = unicode(datetime.datetime.now()).replace(' ','').replace(':','_');
btregressionfilename = "GradientBoostingTreeRegression_" + datestamp + ".txt";
dirfilename = scoredResultDir + btregressionfilename;
predictions.saveAsTextFile(dirfilename)

# PRINT HOW MUCH TIME IT TOOK TO RUN THE CELL
timeend = datetime.datetime.now()
timedelta = round((timeend-timestart).total_seconds(), 2)
print "Time taken to execute above cell: " + str(timedelta) + " seconds";
```

OUTPUT:

Time taken to execute above cell: 14.6 seconds

Clean up objects from memory and print scored file locations

```
# UNPERSIST OBJECTS CACHED IN MEMORY
taxi_df_test_cleaned.unpersist()
indexedTESTbinary.unpersist();
oneHotTESTbinary.unpersist();
indexedTESTreg.unpersist();
oneHotTESTreg.unpersist();
oneHotTESTregScaled.unpersist();

# PRINT OUT PATH TO SCORED OUTPUT FILES
print "logisticRegFileLoc: " + logisticregressionfilename;
print "linearRegFileLoc: " + linearregressionfilename;
print "randomForestClassificationFileLoc: " + rfclassificationfilename;
print "randomForestRegFileLoc: " + rfregressionfilename;
print "BoostedTreeClassificationFileLoc: " + btclassificationfilename;
print "BoostedTreeRegressionFileLoc: " + btregressionfilename;
```

OUTPUT:

logisticRegFileLoc: LogisticRegressionWithLBFGS_2016-05-0317_22_38.953814.txt

linearRegFileLoc: LinearRegressionWithSGD_2016-05-0317_22_58.878949

randomForestClassificationFileLoc: RandomForestClassification_2016-05-0317_23_15.939247.txt

randomForestRegFileLoc: RandomForestRegression_2016-05-0317_23_31.459140.txt

BoostedTreeClassificationFileLoc: GradientBoostingTreeClassification_2016-05-0317_23_49.648334.txt

BoostedTreeRegressionFileLoc: GradientBoostingTreeRegression_2016-05-0317_23_56.860740.txt

Consume Spark Models through a web interface

Spark provides a mechanism to remotely submit batch jobs or interactive queries through a REST interface with a component called Livy. Livy is enabled by default on your HDInsight Spark cluster. For more information on Livy, see: Submit Spark jobs remotely using Livy.

You can use Livy to remotely submit a job that batch scores a file that is stored in an Azure blob and then writes the results to another blob. To do this, you upload the Python script from GitHub to the blob of the Spark cluster. You can use a tool like **Microsoft Azure Storage Explorer** or **AzCopy** to copy the script to the cluster blob. In our case we uploaded the script to ***wasb:///example/python/ConsumeGBNYCReg.py***.

> **NOTE**
> The access keys that you need can be found on the portal for the storage account associated with the Spark cluster.

Once uploaded to this location, this script runs within the Spark cluster in a distributed context. It loads the model and runs predictions on input files based on the model.

You can invoke this script remotely by making a simple HTTPS/REST request on Livy. Here is a curl command to construct the HTTP request to invoke the Python script remotely. Replace CLUSTERLOGIN, CLUSTERPASSWORD, CLUSTERNAME with the appropriate values for your Spark cluster.

```
# CURL COMMAND TO INVOKE PYTHON SCRIPT WITH HTTP REQUEST

curl -k --user "CLUSTERLOGIN:CLUSTERPASSWORD" -X POST --data "{\"file\":
\"wasb:///example/python/ConsumeGBNYCReg.py\"}" -H "Content-Type: application/json"
https://CLUSTERNAME.azurehdinsight.net/livy/batches
```

You can use any language on the remote system to invoke the Spark job through Livy by making a simple HTTPS call with Basic Authentication.

> **NOTE**
> It would be convenient to use the Python Requests library when making this HTTP call, but it is not currently installed by default in Azure Functions. So older HTTP libraries are used instead.

Here is the Python code for the HTTP call:

```
#MAKE AN HTTPS CALL ON LIVY.

import os

# OLDER HTTP LIBRARIES USED HERE INSTEAD OF THE REQUEST LIBRARY AS THEY ARE AVAILABLE BY DEFAULT
import httplib, urllib, base64

# REPLACE VALUE WITH ONES FOR YOUR SPARK CLUSTER
host = '<spark cluster name>.azurehdinsight.net:443'
username='<username>'
password='<password>'

#AUTHORIZATION
conn = httplib.HTTPSConnection(host)
auth = base64.encodestring('%s:%s' % (username, password)).replace('\n', '')
headers = {'Content-Type': 'application/json', 'Authorization': 'Basic %s' % auth}

# SPECIFY THE PYTHON SCRIPT TO RUN ON THE SPARK CLUSTER
# IN THE FILE PARAMETER OF THE JSON POST REQUEST BODY
r=conn.request("POST", '/livy/batches', '{"file": "wasb:///example/python/ConsumeGBNYCReg.py"}', headers )
response = conn.getresponse().read()
print(response)
conn.close()
```

You can also add this Python code to Azure Functions to trigger a Spark job submission that scores a blob based on various events like a timer, creation, or update of a blob.

If you prefer a code free client experience, use the Azure Logic Apps to invoke the Spark batch scoring by defining an HTTP action on the **Logic Apps Designer** and setting its parameters.

- From Azure portal, create a new Logic App by selecting **+New** -> **Web + Mobile** -> **Logic App**.
- To bring up the **Logic Apps Designer**, enter the name of the Logic App and App Service Plan.
- Select an HTTP action and enter the parameters shown in the following figure:

What's next?

Cross-validation and hyperparameter sweeping: See Advanced data exploration and modeling with Spark on how models can be trained using cross-validation and hyper-parameter sweeping.

HDInsight Hadoop data science walkthroughs using Hive on Azure

11/19/2021 • 2 minutes to read • Edit Online

These walkthroughs use Hive with an HDInsight Hadoop cluster to do predictive analytics. They follow the steps outlined in the Team Data Science Process. For an overview of the Team Data Science Process, see Data Science Process. For an introduction to Azure HDInsight, see Introduction to Azure HDInsight, the Hadoop technology stack, and Hadoop clusters.

Additional data science walkthroughs that execute the Team Data Science Process are grouped by the **platform** that they use. See Walkthroughs executing the Team Data Science Process for an itemization of these examples.

Predict taxi tips using Hive with HDInsight Hadoop

The Use HDInsight Hadoop clusters walkthrough uses data from New York taxis to predict:

- Whether a tip is paid
- The distribution of tip amounts

The scenario is implemented using Hive with an Azure HDInsight Hadoop cluster. You learn how to store, explore, and feature engineer data from a publicly available NYC taxi trip and fare dataset. You also use Azure Machine Learning to build and deploy the models.

Predict advertisement clicks using Hive with HDInsight Hadoop

The Use Azure HDInsight Hadoop Clusters on a 1-TB dataset walkthrough uses a publicly available Criteo click dataset to predict whether a tip is paid and the expected amounts. The scenario is implemented using Hive with an Azure HDInsight Hadoop cluster to store, explore, feature engineer, and down sample data. It uses Azure Machine Learning to build, train, and score a binary classification model predicting whether a user clicks on an advertisement. The walkthrough concludes showing how to publish one of these models as a Web service.

Next steps

For a discussion of the key components that comprise the Team Data Science Process, see Team Data Science Process overview.

For a discussion of the Team Data Science Process lifecycle that you can use to structure your data science projects, see Team Data Science Process lifecycle. The lifecycle outlines the steps, from start to finish, that projects usually follow when they are executed.

Azure Data Lake data science walkthroughs using U-SQL

11/19/2021 • 2 minutes to read • Edit Online

These walkthroughs use U-SQL with Azure Data Lake to do predictive analytics. They follow the steps outlined in the Team Data Science Process. For an overview of the Team Data Science Process, see Data Science Process. For an introduction to Azure Data Lake, see Overview of Azure Data Lake Store.

Additional data science walkthroughs that execute the Team Data Science Process are grouped by the **platform** that they use. See Walkthroughs executing the Team Data Science Process for an itemization of these examples.

Predict taxi tips using U-SQL with Azure Data Lake

The Use Azure Data Lake for data science walkthrough shows how to use Azure Data Lake to do data exploration and binary classification tasks. The data are a sample of the NYC taxi dataset. The task is predicting whether or not a tip is paid by a customer.

Next steps

For an overview of the Team Data Science Process, see Team Data Science Process overview.

For a discussion of the Team Data Science Process lifecycle, see Team Data Science Process lifecycle. This lifecycle outlines the steps that projects usually follow when they are executed.

SQL Server data science walkthroughs using R, Python, and T-SQL

11/19/2021 • 2 minutes to read • Edit Online

These walkthroughs use SQL Server, SQL Server R Services, and SQL Server Python Services to do predictive analytics. R and Python code is deployed in stored procedures. They follow the steps outlined in the Team Data Science Process. For an overview of the Team Data Science Process, see Data Science Process.

Additional data science walkthroughs that execute the Team Data Science Process are grouped by the **platform** that they use. See Walkthroughs executing the Team Data Science Process for an itemization of these examples.

Predict taxi tips using Python and SQL queries with SQL Server

The Use SQL Server walkthrough shows how you build and deploy machine learning classification and regression models. The data are a publicly available NYC taxi trip and fare dataset.

Predict taxi tips using Microsoft R with SQL Server

The Use SQL Server R Services walkthrough shows how to build and deploy an R model to SQL Server. The walkthrough is designed to introduce R developers to R Services (In-Database).

Predict taxi tips using R from T-SQL or stored procedures with SQL Server

The Data science walkthrough for R and SQL Server provides SQL programmers with experience building an advanced analytics solution with Transact-SQL using SQL Server R Services to operationalize an R solution.

Predict taxi tips using Python in SQL Server stored procedures

The Use T-SQL with SQL Server Python Services walkthrough provides SQL programmers with experience building a machine learning solution in SQL Server. It demonstrates how to incorporate Python into an application by adding Python code to stored procedures.

Next steps

For a discussion of the key components that comprise the Team Data Science Process, see Team Data Science Process overview.

For a discussion of the Team Data Science Process lifecycle that you can use to structure your data science projects, see Team Data Science Process lifecycle. The lifecycle outlines the steps, from start to finish, that projects usually follow when they are executed.

Azure Synapse Analytics data science walkthroughs using T-SQL and Python on Azure

11/19/2021 • 2 minutes to read • Edit Online

These walkthroughs use of Azure Synapse Analytics to do predictive analytics. They follow the steps outlined in the Team Data Science Process. For an overview of the Team Data Science Process, see Data Science Process. For an introduction to Azure Synapse Analytics, see What is Azure Synapse Analytics?

Additional data science walkthroughs that execute the Team Data Science Process are grouped by the **platform** that they use. See Walkthroughs executing the Team Data Science Process for an itemization of these examples.

Predict taxi tips using T-SQL and IPython notebooks with Azure Synapse Analytics

The Use Azure Synapse Analytics walkthrough shows you how to build and deploy machine learning classification and regression models using Azure Synapse Analytics. The data are a publicly available NYC taxi trip and fare dataset.

Next steps

For a discussion of the key components that comprise the Team Data Science Process, see Team Data Science Process overview.

For a discussion of the Team Data Science Process lifecycle, see Team Data Science Process lifecycle. This lifecycle outlines the steps, from start to finish, that projects usually follow when they are executed.

Team Data Science Process for data scientists

11/19/2021 • 8 minutes to read • Edit Online

This article provides guidance to a set of objectives that are typically used to implement comprehensive data science solutions with Azure technologies. You are guided through:

- understanding an analytics workload
- using the Team Data Science Process
- using Azure Machine Learning
- the foundations of data transfer and storage
- providing data source documentation
- using tools for analytics processing

These training materials are related to the Team Data Science Process (TDSP) and Microsoft and open-source software and toolkits, which are helpful for envisioning, executing and delivering data science solutions.

Lesson Path

You can use the items in the following table to guide your own self-study. Read the *Description* column to follow the path, click on the *Topic* links for study references, and check your skills using the *Knowledge Check* column.

OBJECTIVE	TOPIC	DESCRIPTION	KNOWLEDGE CHECK
Understand the processes for developing analytic projects	An introduction to the Team Data Science Process	We begin by covering an overview of the Team Data Science Process – the TDSP. This process guides you through each step of an analytics project. Read through each of these sections to learn more about the process and how you can implement it.	Review and download the TDSP Project Structure artifacts to your local machine for your project.
	Agile Development	The Team Data Science Process works well with many different programming methodologies. In this Learning Path, we use Agile software development. Read through the "What is Agile Development?" and "Building Agile Culture" articles, which cover the basics of working with Agile. There are also other references at this site where you can learn more.	Explain Continuous Integration and Continuous Delivery to a colleague.

OBJECTIVE	TOPIC	DESCRIPTION	KNOWLEDGE CHECK
	DevOps for Data Science	Developer Operations (DevOps) involves people, processes, and platforms you can use to work through a project and integrate your solution into an organization's standard IT. This integration is essential for adoption, safety, and security. In this online course, you learn about DevOps practices as well as understand some of the toolchain options you have.	Prepare a 30-minute presentation to a technical audience on how DevOps is essential for analytics projects.
Understand the Technologies for Data Storage and Processing	Microsoft Business Analytics and AI	We focus on a few technologies in this Learning Path that you can use to create an analytics solution, but Microsoft has many more. To understand the options you have, it's important to review the platforms and features available in Microsoft Azure, the Azure Stack, and on-premises options. Review this resource to learn the various tools you have available to answer analytics question.	Download and review the presentation materials from this workshop.
Setup and Configure your training, development, and production environments	Microsoft Azure	Now let's create an account in Microsoft Azure for training and learn how to create development and test environments. These free training resources get you started. Complete the "Beginner" and "Intermediate" paths.	If you do not have an Azure Account, create one. Log in to the Microsoft Azure portal and create one Resource Group for training.
	The Microsoft Azure Command-Line Interface (CLI)	There are multiple ways of working with Microsoft Azure – from graphical tools like VSCode and Visual Studio, to Web interfaces such as the Azure portal, and from the command line, such as Azure PowerShell commands and functions. In this article, we cover the Command-Line Interface (CLI), which you can use locally on your workstation, in Windows and other Operating Systems, as well as in the Azure portal.	Set your default subscription with the Azure CLI.

OBJECTIVE	TOPIC	DESCRIPTION	KNOWLEDGE CHECK
	Microsoft Azure Storage	You need a place to store your data. In this article, you learn about Microsoft Azure's storage options, how to create a storage account, and how to copy or move data to the cloud. Read through this introduction to learn more.	Create a Storage Account in your training Resource Group, create a container for a Blob object, and upload and download data.
	Microsoft Azure Active Directory	Microsoft Azure Active Directory (AAD) forms the basis of securing your application. In this article, you learn more about accounts, rights, and permissions. Active Directory and security are complex topics, so just read through this resource to understand the fundamentals.	Add one user to Azure Active Directory. NOTE: You may not have permissions for this action if you are not the administrator for the subscription. If that's the case, simply review this tutorial to learn more.
	The Microsoft Azure Data Science Virtual Machine	You can install the tools for working with Data Science locally on multiple operating systems. But the Microsoft Azure Data Science Virtual Machine (DSVM) contains all of the tools you need and plenty of project samples to work with. In this article, you learn more about the DVSM and how to work through its examples. This resource explains the Data Science Virtual Machine, how you can create one, and a few options for developing code with it. It also contains all the software you need to complete this learning path – so make sure you complete the Knowledge Path for this topic.	Create a Data Science Virtual Machine and work through at least one lab.
Install and Understand the tools and technologies for working with Data Science solutions			

OBJECTIVE	TOPIC	DESCRIPTION	KNOWLEDGE CHECK
Working with git	To follow our DevOps process with the TDSP, we need to have a version-control system. Microsoft Azure Machine Learning uses git, a popular open-source distributed repository system. In this article, you learn more about how to install, configure, and work with git and a central repository – GitHub.	Clone this GitHub project for your learning path project structure.	
	VSCode	VSCode is a cross-platform Integrated Development Environment (IDE) that you can use with multiple languages and Azure tools. You can use this single environment to create your entire solution. Watch these introductory videos to get started.	Install VSCode, and work through the VS Code features in the Interactive Editor Playground.
	Programming with Python	In this solution we use Python, one of the most popular languages in Data Science. This article covers the basics of writing analytic code with Python, and resources to learn more. Work through sections 1-9 of this reference, then check your knowledge.	Add one entity to an Azure Table using Python.
	Working with Notebooks	Notebooks are a way of introducing text and code in the same document. Azure Machine Learning work with Notebooks, so it is beneficial to understand how to use them. Read through this tutorial and give it a try in the Knowledge Check section.	Open this page, and click on the "Welcome to Python.ipynb" link. Work through the examples on that page.
	Machine Learning		

OBJECTIVE	TOPIC	DESCRIPTION	KNOWLEDGE CHECK
Creating advanced Analytic solutions involves working with data, using Machine Learning, which also forms the basis of working with Artificial Intelligence and Deep Learning. This course teaches you more about Machine Learning. For a comprehensive course on Data Science, check out this certification.	Locate a resource on Machine Learning Algorithms. (Hint: Search on "azure machine learning algorithm cheat sheet")		
	scikit-learn	The scikit-learn set of tools allows you to perform data science tasks in Python. We use this framework in our solution. This article covers the basics and explains where you can learn more.	Using the Iris dataset, persist an SVM model using Pickle.
	Working with Docker	Docker is a distributed platform used to build, ship, and run applications, and is used frequently in Azure Machine Learning. This article covers the basics of this technology and explains where you can go to learn more.	Open Visual Studio Code, and install the Docker Extension. Create a simple Node Docker container.
	HDInsight	HDInsight is the Hadoop open-source infrastructure, available as a service in Microsoft Azure. Your Machine Learning algorithms may involve large sets of data, and HDInsight has the ability to store, transfer and process data at large scale. This article covers working with HDInsight.	Create a small HDInsight cluster. Use HiveQL statements to project columns onto an /example/data/sample.log file. Alternatively, you can complete this knowledge check on your local system.

OBJECTIVE	TOPIC	DESCRIPTION	KNOWLEDGE CHECK
Create a Data Processing Flow from Business Requirements	Determining the Question, following the TDSP	With the development environment installed and configured, and the understanding of the technologies and processes in place, it's time to put everything together using the TDSP to perform an analysis. We need to start by defining the question, selecting the data sources, and the rest of the steps in the Team Data Science Process. Keep in mind the DevOps process as we work through this process. In this article, you learn how to take the requirements from your organization and create a data flow map through your application to define your solution using the Team Data Science Process.	
Locate a resource on "The 5 data science questions" and describe one question your organization might have in these areas. Which algorithms should you focus on for that question?			
Use Azure Machine Learning to create a predictive solution	Azure Machine Learning	Microsoft Azure Machine Learning uses AI for data wrangling and feature engineering, manages experiments, and tracks model runs. All of this works in a single environment and most functions can run locally or in Azure. You can use the PyTorch, TensorFlow, and other frameworks to create your experiments. In this article, we focus on a complete example of this process, using everything you've learned so far.	

OBJECTIVE	TOPIC	DESCRIPTION	KNOWLEDGE CHECK
Use Power BI to visualize results	Power BI	Power BI is Microsoft's data visualization tool. It is available on multiple platforms from Web to mobile devices and desktop computers. In this article, you learn how to work with the output of the solution you've created by accessing the results from Azure storage and creating visualizations using Power BI.	Complete this tutorial on Power BI. Then connect Power BI to the Blob CSV created in an experiment run.
Monitor your Solution	Application Insights	There are multiple tools you can use to monitor your end solution. Azure Application Insights makes it easy to integrate built-in monitoring into your solution.	Set up Application Insights to monitor an Application.
	Azure Monitor logs	Another method to monitor your application is to integrate it into your DevOps process. The Azure Monitor logs system provides a rich set of features to help you watch your analytic solutions after you deploy them.	Complete this tutorial on using Azure Monitor logs.
Complete this Learning Path		Congratulations! You've completed this learning path.	

Next steps

Team Data Science Process for Developer Operations This article explores the Developer Operations (DevOps) functions that are specific to an Advanced Analytics and Cognitive Services solution implementation.

Team Data Science Process for Developer Operations

11/19/2021 • 9 minutes to read • Edit Online

This article explores the Developer Operations (DevOps) functions that are specific to an Advanced Analytics and Cognitive Services solution implementation. These training materials implement the Team Data Science Process (TDSP) and Microsoft and open-source software and toolkits, helpful for envisioning, executing and delivering data science solutions. It references topics that cover the DevOps Toolchain that is specific to Data Science and AI projects and solutions.

Lesson Path

The following table provides level-based guidance to help complete the DevOps objectives for implementing data science solutions on Azure.

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
Understand Advanced Analytics	The Team Data Science Process Lifecycle	This technical walkthrough describes the Team Data Science Process	Data Science	Intermediate	General technology background, familiarity with data solutions, Familiarity with IT projects and solution implementation
Understand the Microsoft Azure Platform for Advanced Analytics	Information Management	This reference gives and overview of Azure Data Factory to build pipelines for analytics data solutions	Microsoft Azure Data Factory	Experienced	General technology background, familiarity with data solutions, Familiarity with IT projects and solution implementation
		This reference covers an overview of the Azure Data Catalog which you can use to document and manage metadata on your data sources	Microsoft Azure Data Catalog	Intermediate	General technology background, familiarity with data solutions, familiarity with Relational Database Management Systems (RDBMS) and NoSQL data sources

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
		This reference covers an overview of the Azure Event Hubs system and how you and use it to ingest data into your solution	Azure Event Hubs	Intermediate	General technology background, familiarity with data solutions, familiarity with Relational Database Management Systems (RDBMS) and NoSQL data sources, familiarity with the Internet of Things (IoT) terminology and use
	Big Data Stores	This reference covers an overview of using the Azure Synapse Analytics to store and process large amounts of data	Azure Synapse Analytics	Experienced	General technology background, familiarity with data solutions, familiarity with Relational Database Management Systems (RDBMS) and NoSQL data sources, familiarity with HDFS terminology and use
		This reference covers an overview of using Azure Data Lake to capture data of any size, type, and ingestion speed in one single place for operational and exploratory analytics	Azure Data Lake Store	Intermediate	General technology background, familiarity with data solutions, familiarity with NoSQL data sources, familiarity with HDFS

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
	Machine learning and analytics	This reference covers an introduction to machine learning, predictive analytics, and Artificial Intelligence systems	Azure Machine Learning	Intermediate	General technology background, familiarity with data solutions, familiarity with Data Science terms, familiarity with Machine Learning and artificial intelligence terms
		This article provides an introduction to Azure HDInsight, a cloud distribution of the Hadoop technology stack. It also covers what a Hadoop cluster is and when you would use it	Azure HDInsight	Intermediate	General technology background, familiarity with data solutions, familiarity with NoSQL data sources
		This reference covers an overview of the Azure Data Lake Analytics job service	Azure Data Lake Analytics	Intermediate	General technology background, familiarity with data solutions, familiarity with NoSQL data sources
		This overview covers using Azure Stream Analytics as a fully-managed event-processing engine to up real-time analytic computations on streaming data	Azure Stream Analytics	Intermediate	General technology background, familiarity with data solutions, familiarity with structured and unstructured data concepts
	Intelligence	This reference covers an overview of the available Cognitive Services (such as vision, text, and search) and how to get started using them	Cognitive Services	Experienced	General technology background, familiarity with data solutions, software development

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
		This reference covers and introduction to the Microsoft Bot Framework and how to get started using it	Bot Framework	Experienced	General technology background, familiarity with data solutions
	Visualization	This self-paced, online course covers the Power BI system, and how to create and publish reports	Microsoft Power BI	Beginner	General technology background, familiarity with data solutions
	Solutions	This resource page covers multiple applications you can review, test and implement to see a complete solution from start to finish	Microsoft Azure, Azure Machine Learning, Cognitive Services, Microsoft R, Azure Cognitive Search, Python, Azure Data Factory, Power BI, Azure Document DB, Application Insights, Azure SQL DB, Azure Synapse Analytics, Microsoft SQL Server, Azure Data Lake, Cognitive Services, Bot Framework, Azure Batch,	Intermediate	General technology background, familiarity with data solutions
Understand and Implement DevOps processes	What is DevOps?	This article explains the fundamentals of DevOps and helps explain how they map to DevOps practices	DevOps, Microsoft Azure Platform, Azure DevOps	Intermediate	Familiarity with Agile and other Development Frameworks, IT Operations Familiarity
Use the DevOps Toolchain for Data Science	Configure	This reference covers the basics of choosing the proper visualization in Visio to communicate your project design	Visio	Intermediate	General technology background, familiarity with data solutions

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
		This reference describes the Azure Resource Manager, terms, and serves as the primary root source for samples, getting started, and other references	Azure Resource Manager, Azure PowerShell, Azure CLI	Intermediate	General technology background, familiarity with data solutions
		This reference explains the Azure Data Science Virtual Machines for Linux and Windows	Data Science Virtual Machine	Experienced	Familiarity with Data Science Workloads, Linux
		This walkthrough explains configuring Azure cloud service roles with Visual Studio - pay close attention to the connection strings specifically for storage accounts	Visual Studio	Intermediate	Software Development
		This series teaches you how to use Microsoft Project to schedule time, resources and goals for an Advanced Analytics project	Microsoft Project	Intermediate	Understand Project Management Fundamentals
		This Microsoft Project template provides a time, resources and goals tracking for an Advanced Analytics project	Microsoft Project	Intermediate	Understand Project Management Fundamentals
		This Azure Data Catalog tutorial describes a system of registration and discovery for enterprise data assets	Azure Data Catalog	Beginner	Familiarity with Data Sources and Structures

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
		This Microsoft Virtual Academy course explains how to set up Dev-Test with Visual Studio Codespace and Microsoft Azure	Visual Studio Codespace	Experienced	Software Development, familiarity with Dev/Test environments
		This Management Pack download for Microsoft System Center contains a Guidelines Document to assist in working with Azure assets	System Center	Intermediate	Experience with System Center for IT Management
		This document is intended for developer and operations teams to understand the benefits of PowerShell Desired State Configuration	PowerShell DSC	Intermediate	Experience with PowerShell coding, enterprise architectures, scripting
	Code	This download also contains documentation on using Visual Studio Codespace Code for creating Data Science and AI applications	Visual Studio Codespace	Intermediate	Software Development
		This getting started site teaches you about DevOps and Visual Studio	Visual Studio	Beginner	Software Development
		You can write code directly from the Azure portal using the App Service Editor. Learn more at this resource about Continuous Integration with this tool	Azure portal	Highly Experienced	Data Science background - but read this anyway

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
		This resource explains how to code and create Predictive Analytics experiments using the web-based Azure Machine Learning Studio (classic) tool	Azure Machine Learning Studio (classic)	Experienced	Software Development
		This reference contains a list and a study link to all of the development tools on the Data Science Virtual Machine in Azure	Data Science Virtual Machine	Experienced	Software Development, Data Science
		Read and understand each of the references in this Azure Security Trust Center for Security, Privacy, and Compliance - VERY important	Azure Security	Intermediate	System Architecture Experience, Security Development experience
	Build	This course teaches you about enabling DevOps Practices with Visual Studio Codespace Build	Visual Studio Codespace	Experienced	Software Development, Familiarity with an SDLC
		This reference explains compiling and building using Visual Studio	Visual Studio	Intermediate	Software Development, Familiarity with an SDLC
		This reference explains how to orchestrate processes such as software builds with Runbooks	System Center	Experienced	Experience with System Center Orchestrator

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
	Test	Use this reference to understand how to use Visual Studio Codespace for Test Case Management	Visual Studio Codespace	Experienced	Software Development, Familiarity with an SDLC
		Use this previous reference for Runbooks to automate tests using System Center	System Center	Experienced	Experience with System Center Orchestrator
		As part of not only testing but development, you should build in Security. The Microsoft SDL Threat Modeling Tool can help in all phases. Learn more and download it here	Threat Monitoring Tool	Experienced	Familiarity with security concepts, software development
		This article explains how to use the Microsoft Attack Surface Analyzer to test your Advanced Analytics solution	Attack Surface Analyzer	Experienced	Familiarity with security concepts, software development
	Package	This reference explains the concepts of working with Packages in TFS and Visual Studio Codespace	Visual Studio Codespace	Experienced	Software development, familiarity with an SDLC
		Use this previous reference for Runbooks to automate packaging using System Center	System Center	Experienced	Experience with System Center Orchestrator

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
		This reference explains how to create a data pipeline for your solution, which you can save as a JSON template as a "package"	Azure Data Factory	Intermediate	General computing background, data project experience
		This topic describes the structure of an Azure Resource Manager template	Azure Resource Manager	Intermediate	Familiarity with the Microsoft Azure Platform
		DSC is a management platform in PowerShell that enables you to manage your IT and development infrastructure with configuration as code, saved as a package. This reference is an overview for that topic	PowerShell Desired State Configuration	Intermediate	PowerShell coding, familiarity with enterprise architectures, scripting
	Release	This head-reference article contains concepts for build, test, and release for CI/CD environments	Visual Studio Codespace	Experienced	Software development, familiarity with CI/CD environments, familiarity with an SDLC
		Use this previous reference for Runbooks to automate release management using System Center	System Center	Experienced	Experience with System Center Orchestrator

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
		This article helps you determine the best option to deploy the files for your web app, mobile app backend, or API app to Azure App Service, and then guides you to appropriate resources with instructions specific to your preferred option	Microsoft Azure Deployment	Intermediate	Software development, experience with the Microsoft Azure platform
	Monitor	This reference explains Application Insights and how you can add it to your Advanced Analytics Solutions	Application Insights	Intermediate	Software Development, familiarity with the Microsoft Azure platform
		This topic explains basic concepts about Operations Manager for the administrator who manages the Operations Manager infrastructure and the operator who monitors and supports the Advanced Analytics Solution	System Center	Experienced	Familiarity with enterprise monitoring, System Center Operations Manager
		This blog entry explains how to use the Azure Data Factory to monitor and manage the Advanced Analytics pipeline	Azure Data Factory	Intermediate	Familiarity with Azure Data Factory
		This video shows how to monitor a log with Azure Monitor logs	Azure Logs, PowerShell	Experienced	Familiarity with the Azure Platform

OBJECTIVE	TOPIC	RESOURCE	TECHNOLOGIES	LEVEL	PREREQUISITES
Understand how to use Open Source Tools with DevOps on Azure	Open Source DevOps Tools and Azure	This reference page contains two videos and a whitepaper on using Chef with Azure deployments	Chef	Experienced	Familiarity with the Azure Platform, Familiarity with DevOps
		This site has a toolchain selection path	DevOps, Microsoft Azure Platform, Azure DevOps, Open Source Software	Experienced	Used an SDLC, familiarity with Agile and other Development Frameworks, IT Operations Familiarity
		This tutorial automates the build and test phase of application development using a continuous integration and deployment CI/CD pipeline	Jenkins	Experienced	Familiarity with the Azure Platform, Familiarity with DevOps, Familiarity with Jenkins
		This contains an overview of working with Docker and Azure as well as additional references for implementation for Data Science applications	Docker	Intermediate	Familiarity with the Azure Platform, Familiarity with Server Operating Systems
		This installation and explanation explains how to use Visual Studio Code with Azure assets	VSCODE	Intermediate	Software Development, familiarity with the Microsoft Azure Platform
		This blog entry explains how to use R Studio with Microsoft R	R Studio	Intermediate	R Language experience
		This blog entry shows how to use continuous integration with Azure and GitHub	Git, GitHub	Intermediate	Software Development

Next steps

Team Data Science Process for data scientists This article provides guidance for implementing data science solutions with Azure.

Set up data science environments for use in the Team Data Science Process

11/19/2021 • 2 minutes to read • Edit Online

The Team Data Science Process uses various data science environments for the storage, processing, and analysis of data. They include Azure Blob Storage, several types of Azure virtual machines, HDInsight (Hadoop) clusters, and Machine Learning workspaces. The decision about which environment to use depends on the type and quantity of data to be modeled and the target destination for that data in the cloud.

- For guidance on questions to consider when making this decision, see Plan Your Machine Learning Studio (classic) Data Science Environment.
- For a catalog of some of the scenarios you might encounter when doing advanced analytics, see Scenarios for the Team Data Science Process

The following articles describe how to set up the various data science environments used by the Team Data Science Process.

- Azure storage-account
- HDInsight (Hadoop) cluster
- Machine Learning Studio (classic) workspace

The **Microsoft Data Science Virtual Machine (DSVM)** is also available as an Azure virtual machine (VM) image. This VM is pre-installed and configured with several popular tools that are commonly used for data analytics and machine learning. The DSVM is available on both Windows and Linux. For more information, see Introduction to the cloud-based Data Science Virtual Machine for Linux and Windows.

Learn how to create:

- Windows DSVM
- Ubuntu DSVM
- CentOS DSVM

Platforms and tools for data science projects

11/19/2021 • 9 minutes to read • Edit Online

Microsoft provides a full spectrum of analytics resources for both cloud or on-premises platforms. They can be deployed to make the execution of your data science projects efficient and scalable. Guidance for teams implementing data science projects in a trackable, version controlled, and collaborative way is provided by the Team Data Science Process (TDSP). For an outline of the personnel roles, and their associated tasks that are handled by a data science team standardizing on this process, see Team Data Science Process roles and tasks.

The analytics resources available to data science teams using the TDSP include:

- Data Science Virtual Machines (both Windows and Linux CentOS)
- HDInsight Spark Clusters
- Azure Synapse Analytics
- Azure Data Lake
- HDInsight Hive Clusters
- Azure File Storage
- SQL Server 2019 R and Python Services
- Azure Databricks

In this document, we briefly describe the resources and provide links to the tutorials and walkthroughs the TDSP teams have published. They can help you learn how to use them step by step and start using them to build your intelligent applications. More information on these resources is available on their product pages.

Data Science Virtual Machine (DSVM)

The data science virtual machine offered on both Windows and Linux by Microsoft, contains popular tools for data science modeling and development activities. It includes tools such as:

- Microsoft R Server Developer Edition
- Anaconda Python distribution
- Jupyter notebooks for Python and R
- Visual Studio Community Edition with Python and R Tools on Windows / Eclipse on Linux
- Power BI desktop for Windows
- SQL Server 2016 Developer Edition on Windows / Postgres on Linux

It also includes **ML and AI tools** like xgboost, mxnet, and Vowpal Wabbit.

Currently DSVM is available in **Windows** and **Linux CentOS** operating systems. Choose the size of your DSVM (number of CPU cores and the amount of memory) based on the needs of the data science projects that you are planning to execute on it.

For more information on Windows edition of DSVM, see Microsoft Data Science Virtual Machine on the Azure Marketplace. For the Linux edition of the DSVM, see Linux Data Science Virtual Machine.

To learn how to execute some of the common data science tasks on the DSVM efficiently, see 10 things you can do on the Data science Virtual Machine

Azure HDInsight Spark clusters

Apache Spark is an open-source parallel processing framework that supports in-memory processing to boost

the performance of big-data analytic applications. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory computation capabilities make it a good choice for iterative algorithms in machine learning and for graph computations. Spark is also compatible with Azure Blob storage (WASB), so your existing data stored in Azure can easily be processed using Spark.

When you create a Spark cluster in HDInsight, you create Azure compute resources with Spark installed and configured. It takes about 10 minutes to create a Spark cluster in HDInsight. Store the data to be processed in Azure Blob storage. For information on using Azure Blob Storage with a cluster, see Use HDFS-compatible Azure Blob storage with Hadoop in HDInsight.

TDSP team from Microsoft has published two end-to-end walkthroughs on how to use Azure HDInsight Spark Clusters to build data science solutions, one using Python and the other Scala. For more information on Azure HDInsight **Spark Clusters**, see Overview: Apache Spark on HDInsight Linux. To learn how to build a data science solution using **Python** on an Azure HDInsight Spark Cluster, see Overview of Data Science using Spark on Azure HDInsight. To learn how to build a data science solution using **Scala** on an Azure HDInsight Spark Cluster, see Data Science using Scala and Spark on Azure.

Azure Synapse Analytics

Azure Synapse Analytics allows you to scale compute resources easily and in seconds, without over-provisioning or over-paying. It also offers the unique option to pause the use of compute resources, giving you the freedom to better manage your cloud costs. The ability to deploy scalable compute resources makes it possible to bring all your data into Azure Synapse Analytics. Storage costs are minimal and you can run compute only on the parts of datasets that you want to analyze.

For more information on Azure Synapse Analytics, see the Azure Synapse Analytics website. To learn how to build end-to-end advanced analytics solutions with Azure Synapse Analytics, see The Team Data Science Process in action: using Azure Synapse Analytics.

Azure Data Lake

Azure Data Lake is as an enterprise-wide repository of every type of data collected in a single location, prior to any formal requirements, or schema being imposed. This flexibility allows every type of data to be kept in a data lake, regardless of its size or structure or how fast it is ingested. Organizations can then use Hadoop or advanced analytics to find patterns in these data lakes. Data lakes can also serve as a repository for lower-cost data preparation before curating the data and moving it into a data warehouse.

For more information on Azure Data Lake, see Introducing Azure Data Lake. To learn how to build a scalable end-to-end data science solution with Azure Data Lake, see Scalable Data Science in Azure Data Lake: An end-to-end Walkthrough

Azure HDInsight Hive (Hadoop) clusters

Apache Hive is a data warehouse system for Hadoop, which enables data summarization, querying, and the analysis of data using HiveQL, a query language similar to SQL. Hive can be used to interactively explore your data or to create reusable batch processing jobs.

Hive allows you to project structure on largely unstructured data. After you define the structure, you can use Hive to query that data in a Hadoop cluster without having to use, or even know, Java or MapReduce. HiveQL (the Hive query language) allows you to write queries with statements that are similar to T-SQL.

For data scientists, Hive can run Python User-Defined Functions (UDFs) in Hive queries to process records. This ability extends the capability of Hive queries in data analysis considerably. Specifically, it allows data scientists to conduct scalable feature engineering in languages they are mostly familiar with: the SQL-like HiveQL and Python.

For more information on Azure HDInsight Hive Clusters, see Use Hive and HiveQL with Hadoop in HDInsight. To learn how to build a scalable end-to-end data science solution with Azure HDInsight Hive Clusters, see The Team Data Science Process in action: using HDInsight Hadoop clusters.

Azure File Storage

Azure File Storage is a service that offers file shares in the cloud using the standard Server Message Block (SMB) Protocol. Both SMB 2.1 and SMB 3.0 are supported. With Azure File storage, you can migrate legacy applications that rely on file shares to Azure quickly and without costly rewrites. Applications running in Azure virtual machines or cloud services or from on-premises clients can mount a file share in the cloud, just as a desktop application mounts a typical SMB share. Any number of application components can then mount and access the File storage share simultaneously.

Especially useful for data science projects is the ability to create an Azure file store as the place to share project data with your project team members. Each of them then has access to the same copy of the data in the Azure file storage. They can also use this file storage to share feature sets generated during the execution of the project. If the project is a client engagement, your clients can create an Azure file storage under their own Azure subscription to share the project data and features with you. In this way, the client has full control of the project data assets. For more information on Azure File Storage, see Get started with Azure File storage on Windows and How to use Azure File Storage with Linux.

SQL Server 2019 R and Python Services

R Services (In-database) provides a platform for developing and deploying intelligent applications that can uncover new insights. You can use the rich and powerful R language, including the many packages provided by the R community, to create models and generate predictions from your SQL Server data. Because R Services (In-database) integrates the R language with SQL Server, analytics are kept close to the data, which eliminates the costs and security risks associated with moving data.

R Services (In-database) supports the open source R language with a comprehensive set of SQL Server tools and technologies. They offer superior performance, security, reliability, and manageability. You can deploy R solutions using convenient and familiar tools. Your production applications can call the R runtime and retrieve predictions and visuals using Transact-SQL. You also use the ScaleR libraries to improve the scale and performance of your R solutions. For more information, see SQL Server R Services.

The TDSP team from Microsoft has published two end-to-end walkthroughs that show how to build data science solutions in SQL Server 2016 R Services: one for R programmers and one for SQL developers. For **R Programmers**, see Data Science End-to-End Walkthrough. For **SQL Developers**, see In-Database Advanced Analytics for SQL Developers (Tutorial).

Appendix: Tools to set up data science projects

Install Git Credential Manager on Windows

If you are following the TDSP on **Windows**, you need to install the **Git Credential Manager (GCM)** to communicate with the Git repositories. To install GCM, you first need to install **Chocolaty**. To install Chocolaty and the GCM, run the following commands in Windows PowerShell as an **Administrator**:

```
iwr https://chocolatey.org/install.ps1 -UseBasicParsing | iex
choco install git-credential-manager-for-windows -y
```

Install Git on Linux (CentOS) machines

Run the following bash command to install Git on Linux (CentOS) machines:

```
sudo yum install git
```

Generate public SSH key on Linux (CentOS) machines

If you are using Linux (CentOS) machines to run the git commands, you need to add the public SSH key of your machine to your Azure DevOps Services, so that this machine is recognized by the Azure DevOps Services. First, you need to generate a public SSH key and add the key to SSH public keys in your Azure DevOps Services security setting page.

1. To generate the SSH key, run the following two commands:

```
ssh-keygen
cat .ssh/id_rsa.pub
```

2. Copy the entire ssh key including *ssh-rsa*.

3. Log in to your Azure DevOps Services.

4. Click **<Your Name>** at the top-right corner of the page and click **security**.

5. Click **SSH public keys**, and click **+Add**.

6. Paste the ssh key copied into the text box and save.

Next steps

Full end-to-end walkthroughs that demonstrate all the steps in the process for **specific scenarios** are also provided. They are listed and linked with thumbnail descriptions in the Example walkthroughs topic. They illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

For examples that show how to execute steps in the Team Data Science Process by using Azure Machine Learning Studio (classic), see the With Azure ML learning path.

Identify scenarios and plan for advanced analytics data processing

11/19/2021 • 4 minutes to read • Edit Online

What resources are required for you to create an environment that can perform advanced analytics processing on a dataset? This article suggests a series of questions to ask that can help identify tasks and resources relevant your scenario.

To learn about the order of high-level steps for predictive analytics, see What is the Team Data Science Process (TDSP). Each step requires specific resources for the tasks relevant to your particular scenario.

Answer key questions in the following areas to identify your scenario:

- data logistics
- data characteristics
- dataset quality
- preferred tools and languages

Logistic questions: data locations and movement

The logistic questions cover the following items:

- data source location
- target destination in Azure
- requirements for moving the data, including the schedule, amount, and resources involved

You may need to move the data several times during the analytics process. A common scenario is to move local data into some form of storage on Azure and then into Machine Learning Studio.

What is your data source?

Is your data local or in the cloud? Possible locations include:

- a publicly available HTTP address
- a local or network file location
- a SQL Server database
- an Azure Storage container

What is the Azure destination?

Where does your data need to be for processing or modeling?

- Azure Blob Storage
- SQL Azure databases
- SQL Server on Azure VM
- HDInsight (Hadoop on Azure) or Hive tables
- Azure Machine Learning
- Mountable Azure virtual hard disks

How are you going to move the data?

For procedures and resources to ingest or load data into a variety of different storage and processing environments, see:

- Load data into storage environments for analytics
- Import your training data into Machine Learning Studio (classic) from various data sources

Does the data need to be moved on a regular schedule or modified during migration?

Consider using Azure Data Factory (ADF) when data needs to be continually migrated. ADF can be helpful for:

- a hybrid scenario that involves both on-premises and cloud resources
- a scenario where the data is transacted, modified, or changed by business logic in the course of being migrated

For more information, see Move data from a SQL Server database to SQL Azure with Azure Data Factory.

How much of the data is to be moved to Azure?

Large datasets may exceed the storage capacity of certain environments. For an example, see the discussion of size limits for Machine Learning Studio (classic) in the next section. In such cases, you might use a sample of the data during the analysis. For details of how to down-sample a dataset in various Azure environments, see Sample data in the Team Data Science Process.

Data characteristics questions: type, format, and size

These questions are key to planning your storage and processing environments. They will help you choose the appropriate scenario for your data type and understand any restrictions.

What are the data types?

- Numerical
- Categorical
- Strings
- Binary

How is your data formatted?

- Comma-separated (CSV) or tab-separated (TSV) flat files
- Compressed or uncompressed
- Azure blobs
- Hadoop Hive tables
- SQL Server tables

How large is your data?

- Small: Less than 2 GB
- Medium: Greater than 2 GB and less than 10 GB
- Large: Greater than 10 GB

Take the Machine Learning Studio (classic) environment for example:

- For a list of the data formats and types supported by Machine Learning Studio (classic), see Data formats and data types supported section.
- For information on the limitations of other Azure services used in the analytics process, see Azure Subscription and Service Limits, Quotas, and Constraints.

Data quality questions: exploration and pre-processing

What do you know about your data?

Understand the basic characteristics about your data:

- What patterns or trends it exhibits

- What outliers it has
- How many values are missing

This step is important to help you:

- Determine how much pre-processing is needed
- Formulate hypotheses that suggest the most appropriate features or type of analysis
- Formulate plans for additional data collection

Useful techniques for data inspection include descriptive statistics calculation and visualization plots. For details of how to explore a dataset in various Azure environments, see Explore data in the Team Data Science Process.

Does the data require preprocessing or cleaning?

You might need to preprocess and clean your data before you can use the dataset effectively for machine learning. Raw data is often noisy and unreliable. It might be missing values. Using such data for modeling can produce misleading results. For a description, see Tasks to prepare data for enhanced machine learning.

Tools and languages questions

There are many options for languages, development environments, and tools. Be aware of your needs and preferences.

What languages do you prefer to use for analysis?

- R
- Python
- SQL

What tools should you use for data analysis?

- Microsoft Azure PowerShell - a script language used to administer your Azure resources in a script language
- Azure Machine Learning Studio
- Revolution Analytics
- RStudio
- Python Tools for Visual Studio
- Anaconda
- Jupyter notebooks
- Microsoft Power BI

Identify your advanced analytics scenario

After you have answered the questions in the previous section, you are ready to determine which scenario best fits your case. The sample scenarios are outlined in Scenarios for advanced analytics in Azure Machine Learning.

Next steps

What is the Team Data Science Process (TDSP)?

Load data into storage environments for analytics

11/19/2021 • 2 minutes to read • Edit Online

The Team Data Science Process requires that data be ingested or loaded into the most appropriate way in each stage. Data destinations can include Azure Blob Storage, SQL Azure databases, SQL Server on Azure VM, HDInsight (Hadoop), Azure Synapse Analytics, and Azure Machine Learning.

The following articles describe how to ingest data into various target environments where the data is stored and processed.

- To/From Azure Blob Storage
- To SQL Server on Azure VM
- To Azure SQL Database
- To Hive tables
- To SQL partitioned tables
- From On-premises SQL Server

Technical and business needs, as well as the initial location, format, and size of your data will determine the best data ingestion plan. It is not uncommon for a best plan to have several steps. This sequence of tasks can include, for example, data exploration, pre-processing, cleaning, down-sampling, and model training. Azure Data Factory is a recommended Azure resource to orchestrate data movement and transformation.

Move data to and from Azure Blob storage

11/19/2021 • 2 minutes to read • Edit Online

The Team Data Science Process requires that data be ingested or loaded into a variety of different storage environments to be processed or analyzed in the most appropriate way in each stage of the process.

Different technologies for moving data

The following articles describe how to move data to and from Azure Blob storage using different technologies.

- Azure Storage-Explorer
- AzCopy
- Python
- SSIS

Which method is best for you depends on your scenario. The Scenarios for advanced analytics in Azure Machine Learning article helps you determine the resources you need for a variety of data science workflows used in the advanced analytics process.

> **NOTE**
>
> For a complete introduction to Azure blob storage, refer to Azure Blob Basics and to Azure Blob Service.

Using Azure Data Factory

As an alternative, you can use Azure Data Factory to do the following:

- Create and schedule a pipeline that downloads data from Azure Blob storage.
- Pass it to a published Azure Machine Learning web service.
- Receive the predictive analytics results.
- Upload the results to storage.

For more information, see Create predictive pipelines using Azure Data Factory and Azure Machine Learning.

Prerequisites

This article assumes that you have an Azure subscription, a storage account, and the corresponding storage key for that account. Before uploading/downloading data, you must know your Azure Storage account name and account key.

- To set up an Azure subscription, see Free one-month trial.
- For instructions on creating a storage account and for getting account and key information, see About Azure Storage accounts.

Move data to and from Azure Blob Storage using Azure Storage Explorer

11/19/2021 • 2 minutes to read • Edit Online

Azure Storage Explorer is a free tool from Microsoft that allows you to work with Azure Storage data on Windows, macOS, and Linux. This topic describes how to use it to upload and download data from Azure Blob Storage. The tool can be downloaded from Microsoft Azure Storage Explorer.

This menu links to technologies you can use to move data to and from Azure Blob storage:

> **NOTE**
>
> If you are using VM that was set up with the scripts provided by Data Science Virtual machines in Azure, then Azure Storage Explorer is already installed on the VM.

> **NOTE**
>
> For a complete introduction to Azure Blob Storage, refer to Azure Blob Basics and Azure Blob Service.

Prerequisites

This document assumes that you have an Azure subscription, a storage account, and the corresponding storage key for that account. Before uploading/downloading data, you must know your Azure Storage account name and account key.

- To set up an Azure subscription, see Free one-month trial.
- For instructions on creating a storage account and for getting account and key information, see About Azure Storage accounts. Make a note the access key for your storage account as you need this key to connect to the account with the Azure Storage Explorer tool.
- The Azure Storage Explorer tool can be downloaded from Microsoft Azure Storage Explorer. Accept the defaults during install.

Use Azure Storage Explorer

The following steps document how to upload/download data using Azure Storage Explorer.

1. Launch Microsoft Azure Storage Explorer.
2. To bring up the **Sign in to your account...** wizard, select **Azure account settings** icon, then **Add an account** and enter you credentials.

3. To bring up the **Connect to Azure Storage** wizard, select the **Connect to Azure Storage** icon.

4. Enter the access key from your Azure Storage account on the **Connect to Azure Storage** wizard and then **Next**.

5. Enter storage account name in the **Account name** box and then select **Next**.

6. The storage account added should now be displayed. To create a blob container in a storage account, right-click the **Blob Containers** node in that account, select **Create Blob Container**, and enter a name.
7. To upload data to a container, select the target container and click the **Upload** button.

8. Click on the ... to the right of the **Files** box, select one or multiple files to upload from the file system and click **Upload** to begin uploading the files.

9. To download data, selecting the blob in the corresponding container to download and click **Download**.

Move data to or from Azure Blob Storage using SSIS connectors

11/19/2021 • 3 minutes to read • Edit Online

The SQL Server Integration Services Feature Pack for Azure provides components to connect to Azure, transfer data between Azure and on-premises data sources, and process data stored in Azure.

This menu links to technologies you can use to move data to and from Azure Blob storage:

Once customers have moved on-premises data into the cloud, they can access their data from any Azure service to leverage the full power of the suite of Azure technologies. The data may be subsequently used, for example, in Azure Machine Learning or on an HDInsight cluster.

Examples for using these Azure resources are in the SQL and HDInsight walkthroughs.

For a discussion of canonical scenarios that use SSIS to accomplish business needs common in hybrid data integration scenarios, see Doing more with SQL Server Integration Services Feature Pack for Azure blog.

> **NOTE**
>
> For a complete introduction to Azure blob storage, refer to Azure Blob Basics and to Azure Blob Service.

Prerequisites

To perform the tasks described in this article, you must have an Azure subscription and an Azure Storage account set up. You need the Azure Storage account name and account key to upload or download data.

- To set up an **Azure subscription**, see Free one-month trial.
- For instructions on creating a **storage account** and for getting account and key information, see About Azure Storage accounts.

To use the **SSIS connectors**, you must download:

- **SQL Server 2014 or 2016 Standard (or above)**: Install includes SQL Server Integration Services.
- **Microsoft SQL Server 2014 or 2016 Integration Services Feature Pack for Azure**: These connectors can be downloaded, respectively, from the SQL Server 2014 Integration Services and SQL Server 2016 Integration Services pages.

> **NOTE**
>
> SSIS is installed with SQL Server, but is not included in the Express version. For information on what applications are included in various editions of SQL Server, see SQL Server Editions

For training materials on SSIS, see Hands On Training for SSIS

For information on how to get up-and-running using SISS to build simple extraction, transformation, and load (ETL) packages, see SSIS Tutorial: Creating a Simple ETL Package.

Download NYC Taxi dataset

The example described here use a publicly available dataset -- the NYC Taxi Trips dataset. The dataset consists of

about 173 million taxi rides in NYC in the year 2013. There are two types of data: trip details data and fare data. As there is a file for each month, we have 24 files, each of which is about 2 GB uncompressed.

Upload data to Azure blob storage

To move data using the SSIS feature pack from on-premises to Azure blob storage, we use an instance of the Azure Blob Upload Task, shown here:

The parameters that the task uses are described here:

FIELD	DESCRIPTION
AzureStorageConnection	Specifies an existing Azure Storage Connection Manager or creates a new one that refers to an Azure Storage account that points to where the blob files are hosted.
BlobContainer	Specifies the name of the blob container that holds the uploaded files as blobs.
BlobDirectory	Specifies the blob directory where the uploaded file is stored as a block blob. The blob directory is a virtual hierarchical structure. If the blob already exists, it ia replaced.
LocalDirectory	Specifies the local directory that contains the files to be uploaded.
FileName	Specifies a name filter to select files with the specified name pattern. For example, MySheet*.xls* includes files such as MySheet001.xls and MySheetABC.xlsx
TimeRangeFrom/TimeRangeTo	Specifies a time range filter. Files modified after *TimeRangeFrom* and before *TimeRangeTo* are included.

> **NOTE**
> The **AzureStorageConnection** credentials need to be correct and the **BlobContainer** must exist before the transfer is attempted.

Download data from Azure blob storage

To download data from Azure blob storage to on-premises storage with SSIS, use an instance of the Azure Blob Download Task.

More advanced SSIS-Azure scenarios

The SSIS feature pack allows for more complex flows to be handled by packaging tasks together. For example, the blob data could feed directly into an HDInsight cluster, whose output could be downloaded back to a blob and then to on-premises storage. SSIS can run Hive and Pig jobs on an HDInsight cluster using additional SSIS connectors:

- To run a Hive script on an Azure HDInsight cluster with SSIS, use Azure HDInsight Hive Task.
- To run a Pig script on an Azure HDInsight cluster with SSIS, use Azure HDInsight Pig Task.

Move data to SQL Server on an Azure virtual machine

11/19/2021 • 7 minutes to read • Edit Online

This article outlines the options for moving data either from flat files (CSV or TSV formats) or from an on-premises SQL Server to SQL Server on an Azure virtual machine. These tasks for moving data to the cloud are part of the Team Data Science Process.

For a topic that outlines the options for moving data to an Azure SQL Database for Machine Learning, see Move data to an Azure SQL Database for Azure Machine Learning.

The following table summarizes the options for moving data to SQL Server on an Azure virtual machine.

SOURCE	DESTINATION: SQL SERVER ON AZURE VM
Flat File	1. Command-line bulk copy utility (BCP) 2. Bulk Insert SQL Query 3. Graphical Built-in Utilities in SQL Server
On-Premises SQL Server	1. Deploy a SQL Server Database to a Microsoft Azure VM wizard 2. Export to a flat File 3. SQL Database Migration Wizard 4. Database back up and restore

This document assumes that SQL commands are executed from SQL Server Management Studio or Visual Studio Database Explorer.

> **TIP**
>
> As an alternative, you can use Azure Data Factory to create and schedule a pipeline that will move data to a SQL Server VM on Azure. For more information, see Copy data with Azure Data Factory (Copy Activity).

Prerequisites

This tutorial assumes you have:

- An **Azure subscription**. If you do not have a subscription, you can sign up for a free trial.
- An **Azure storage account**. You will use an Azure storage account for storing the data in this tutorial. If you don't have an Azure storage account, see the Create a storage account article. After you have created the storage account, you will need to obtain the account key used to access the storage. See Manage storage account access keys.
- Provisioned **SQL Server on an Azure VM**. For instructions, see Set up an Azure SQL Server virtual machine as an IPython Notebook server for advanced analytics.
- Installed and configured **Azure PowerShell** locally. For instructions, see How to install and configure Azure PowerShell.

Moving data from a flat file source to SQL Server on an Azure VM

If your data is in a flat file (arranged in a row/column format), it can be moved to SQL Server VM on Azure via

the following methods:

1. Command-line bulk copy utility (BCP)
2. Bulk Insert SQL Query
3. Graphical Built-in Utilities in SQL Server (Import/Export, SSIS)

Command-line bulk copy utility (BCP)

BCP is a command-line utility installed with SQL Server and is one of the quickest ways to move data. It works across all three SQL Server variants (On-premises SQL Server, SQL Azure, and SQL Server VM on Azure).

> **NOTE**
>
> **Where should my data be for BCP?** While it is not required, having files containing source data located on the same machine as the target SQL Server allows for faster transfers (network speed vs local disk IO speed). You can move the flat files containing data to the machine where SQL Server is installed using various file copying tools such as AZCopy, Azure Storage Explorer or windows copy/paste via Remote Desktop Protocol (RDP).

1. Ensure that the database and the tables are created on the target SQL Server database. Here is an example of how to do that using the `Create Database` and `Create Table` commands:

```
CREATE DATABASE <database_name>

CREATE TABLE <tablename>
(
    <columnname1> <datatype> <constraint>,
    <columnname2> <datatype> <constraint>,
    <columnname3> <datatype> <constraint>
)
```

2. Generate the format file that describes the schema for the table by issuing the following command from the command line of the machine where bcp is installed.

```
bcp dbname..tablename format nul -c -x -f exportformatfilename.xml -S servername\sqlinstance -T -t \t
-r \n
```

3. Insert the data into the database using the bcp command, which should work from the command line when SQL Server is installed on same machine:

```
bcp dbname..tablename in datafilename.tsv -f exportformatfilename.xml -S servername\sqlinstancename -U
username -P password -b block_size_to_move_in_single_attempt -t \t -r \n
```

> **Optimizing BCP Inserts** Please refer the following article 'Guidelines for Optimizing Bulk Import' to optimize such inserts.

Parallelizing Inserts for Faster Data Movement

If the data you are moving is large, you can speed up things by simultaneously executing multiple BCP commands in parallel in a PowerShell Script.

> **NOTE**
>
> **Big data Ingestion** To optimize data loading for large and very large datasets, partition your logical and physical database tables using multiple file groups and partition tables. For more information about creating and loading data to partition tables, see Parallel Load SQL Partition Tables.

The following sample PowerShell script demonstrates parallel inserts using bcp:

```
$NO_OF_PARALLEL_JOBS=2

Set-ExecutionPolicy RemoteSigned #set execution policy for the script to execute
# Define what each job does
$ScriptBlock = {
    param($partitionnumber)

    #Explicitly using SQL username password
    bcp database..tablename in datafile_path.csv -F 2 -f format_file_path.xml -U username@servername -S
tcp:servername -P password -b block_size_to_move_in_single_attempt -t "," -r \n -o
path_to_outputfile.$partitionnumber.txt

    #Trusted connection w.o username password (if you are using windows auth and are signed in with that
credentials)
    #bcp database..tablename in datafile_path.csv -o path_to_outputfile.$partitionnumber.txt -h "TABLOCK" -F
2 -f format_file_path.xml  -T -b block_size_to_move_in_single_attempt -t "," -r \n
}

# Background processing of all partitions
for ($i=1; $i -le $NO_OF_PARALLEL_JOBS; $i++)
{
    Write-Debug "Submit loading partition # $i"
    Start-Job $ScriptBlock -Arg $i
}

# Wait for it all to complete
While (Get-Job -State "Running")
{
    Start-Sleep 10
    Get-Job
}

# Getting the information back from the jobs
Get-Job | Receive-Job
Set-ExecutionPolicy Restricted #reset the execution policy
```

Bulk Insert SQL Query

Bulk Insert SQL Query can be used to import data into the database from row/column based files (the supported types are covered in thePrepare Data for Bulk Export or Import (SQL Server)) topic.

Here are some sample commands for Bulk Insert are as below:

1. Analyze your data and set any custom options before importing to make sure that the SQL Server database assumes the same format for any special fields such as dates. Here is an example of how to set the date format as year-month-day (if your data contains the date in year-month-day format):

   ```
   SET DATEFORMAT ymd;
   ```

2. Import data using bulk import statements:

   ```
   BULK INSERT <tablename>
   FROM
   '<datafilename>'
   WITH
   (
       FirstRow = 2,
       FIELDTERMINATOR = ',', --this should be column separator in your data
       ROWTERMINATOR = '\n'   --this should be the row separator in your data
   )
   ```

Built-in Utilities in SQL Server

You can use SQL Server Integration Services (SSIS) to import data into SQL Server VM on Azure from a flat file. SSIS is available in two studio environments. For details, see Integration Services (SSIS) and Studio Environments:

- For details on SQL Server Data Tools, see Microsoft SQL Server Data Tools
- For details on the Import/Export Wizard, see SQL Server Import and Export Wizard

Moving Data from on-premises SQL Server to SQL Server on an Azure VM

You can also use the following migration strategies:

1. Deploy a SQL Server Database to a Microsoft Azure VM wizard
2. Export to Flat File
3. SQL Database Migration Wizard
4. Database back up and restore

We describe each of these options below:

Deploy a SQL Server Database to a Microsoft Azure VM wizard

The **Deploy a SQL Server Database to a Microsoft Azure VM wizard** is a simple and recommended way to move data from an on-premises SQL Server instance to SQL Server on an Azure VM. For detailed steps as well as a discussion of other alternatives, see Migrate a database to SQL Server on an Azure VM.

Export to Flat File

Various methods can be used to bulk export data from an On-Premises SQL Server as documented in the Bulk Import and Export of Data (SQL Server) topic. This document will cover the Bulk Copy Program (BCP) as an example. Once data is exported into a flat file, it can be imported to another SQL server using bulk import.

1. Export the data from on-premises SQL Server to a file using the bcp utility as follows

   ```
   bcp dbname..tablename out datafile.tsv -S servername\sqlinstancename -T -t \t -t \n -c
   ```

2. Create the database and the table on SQL Server VM on Azure using the `create database` and `create table` for the table schema exported in step 1.

3. Create a format file for describing the table schema of the data being exported/imported. Details of the format file are described in Create a Format File (SQL Server).

   Format file generation when running BCP from the SQL Server computer

   ```
   bcp dbname..tablename format nul -c -x -f exportformatfilename.xml -S servername\sqlinstance -T -t \t
   -r \n
   ```

   Format file generation when running BCP remotely against a SQL Server

   ```
   bcp dbname..tablename format nul -c -x -f exportformatfilename.xml -U
   username@servername.database.windows.net -S tcp:servername -P password --t \t -r \n
   ```

4. Use any of the methods described in section Moving Data from File Source to move the data in flat files to a SQL Server.

SQL Database Migration Wizard

Microsoft Data Migration provides a user-friendly way to move data between two SQL server instances. It allows the user to map the data schema between sources and destination tables, choose column types and various other functionalities. It uses bulk copy (BCP) under the covers. A screenshot of the welcome screen for the SQL Database Migration wizard is shown below.

Database back up and restore

SQL Server supports:

1. Database back up and restore functionality (both to a local file or bacpac export to blob) and Data Tier Applications (using bacpac).
2. Ability to directly create SQL Server VMs on Azure with a copied database or copy to an existing database in SQL Database. For more information, see Use the Copy Database Wizard.

A screenshot of the Database back up/restore options from SQL Server Management Studio is shown below.

Resources

Migrate a Database to SQL Server on an Azure VM

SQL Server on Azure Virtual Machines overview

Move data to Azure SQL Database for Azure Machine Learning

11/19/2021 • 3 minutes to read • Edit Online

This article outlines the options for moving data either from flat files (CSV or TSV formats) or from data stored in SQL Server to an Azure SQL Database. These tasks for moving data to the cloud are part of the Team Data Science Process.

For a topic that outlines the options for moving data to SQL Server for Machine Learning, see Move data to SQL Server on an Azure virtual machine.

The following table summarizes the options for moving data to an Azure SQL Database.

SOURCE	DESTINATION: AZURE SQL DATABASE
Flat file (CSV or TSV formatted)	Bulk Insert SQL Query
On-premises SQL Server	1.Export to Flat File 2. SQL Database Migration Wizard 3. Database back up and restore 4. Azure Data Factory

Prerequisites

The procedures outlined here require that you have:

- An **Azure subscription**. If you do not have a subscription, you can sign up for a free trial.
- An **Azure storage account**. You use an Azure storage account for storing the data in this tutorial. If you don't have an Azure storage account, see the Create a storage account article. After you have created the storage account, you need to obtain the account key used to access the storage. See Manage storage account access keys.
- Access to an **Azure SQL Database**. If you must set up an Azure SQL Database, Getting Started with Microsoft Azure SQL Database provides information on how to provision a new instance of an Azure SQL Database.
- Installed and configured **Azure PowerShell** locally. For instructions, see How to install and configure Azure PowerShell.

Data: The migration processes are demonstrated using the NYC Taxi dataset. The NYC Taxi dataset contains information on trip data and fairs and is available on Azure blob storage: NYC Taxi Data. A sample and description of these files are provided in NYC Taxi Trips Dataset Description.

You can either adapt the procedures described here to a set of your own data or follow the steps as described by using the NYC Taxi dataset. To upload the NYC Taxi dataset into your SQL Server database, follow the procedure outlined in Bulk Import Data into SQL Server Database.

Moving data from a flat file source to an Azure SQL Database

Data in flat files (CSV or TSV formatted) can be moved to an Azure SQL Database using a Bulk Insert SQL Query.

Bulk Insert SQL Query

The steps for the procedure using the Bulk Insert SQL Query are similar to the directions for moving data from a flat file source to SQL Server on an Azure VM. For details, see Bulk Insert SQL Query.

Moving Data from SQL Server to an Azure SQL Database

If the source data is stored in SQL Server, there are various possibilities for moving the data to an Azure SQL Database:

1. Export to Flat File
2. SQL Database Migration Wizard
3. Database back up and restore
4. Azure Data Factory

The steps for the first three are similar to those sections in Move data to SQL Server on an Azure virtual machine that cover these same procedures. Links to the appropriate sections in that topic are provided in the following instructions.

Export to Flat File

The steps for this exporting to a flat file are similar to those directions covered in Export to Flat File.

SQL Database Migration Wizard

The steps for using the SQL Database Migration Wizard are similar to those directions covered in SQL Database Migration Wizard.

Database back up and restore

The steps for using database backup and restore are similar to those directions listed in Database backup and restore.

Azure Data Factory

Learn how to move data to an Azure SQL Database with Azure Data Factory (ADF) in this topic, Move data from a SQL Server to SQL Azure with Azure Data Factory. This topic shows how to use ADF to move data from a SQL Server database to an Azure SQL Database via Azure Blob Storage.

Consider using ADF when data needs to be continually migrated with hybrid on-premises and cloud sources. ADF also helps when the data needs transformations, or needs new business logic during migration. ADF allows for the scheduling and monitoring of jobs using simple JSON scripts that manage the movement of data on a periodic basis. ADF also has other capabilities such as support for complex operations.

Create Hive tables and load data from Azure Blob Storage

11/19/2021 • 10 minutes to read • Edit Online

This article presents generic Hive queries that create Hive tables and load data from Azure Blob Storage. Some guidance is also provided on partitioning Hive tables and on using the Optimized Row Columnar (ORC) formatting to improve query performance.

Prerequisites

This article assumes that you have:

- Created an Azure Storage account. If you need instructions, see About Azure Storage accounts.
- Provisioned a customized Hadoop cluster with the HDInsight service. If you need instructions, see Setup Clusters in HDInsight.
- Enabled remote access to the cluster, logged in, and opened the Hadoop Command-Line console. If you need instructions, see Manage Apache Hadoop clusters.

Upload data to Azure Blob Storage

If you created an Azure virtual machine by following the instructions provided in Set up an Azure virtual machine for advanced analytics, this script file should have been downloaded to the *C:\Users\<user name>\Documents\Data Science Scripts* directory on the virtual machine. These Hive queries only require that you provide a data schema and Azure Blob Storage configuration in the appropriate fields to be ready for submission.

We assume that the data for Hive tables is in an **uncompressed** tabular format, and that the data has been uploaded to the default (or to an additional) container of the storage account used by the Hadoop cluster.

If you want to practice on the **NYC Taxi Trip Data**, you need to:

- **download** the 24 NYC Taxi Trip Data files (12 Trip files and 12 Fare files),
- **unzip** all files into .csv files, and then
- **upload** them to the default (or appropriate container) of the Azure Storage account; options for such an account appear at Use Azure Storage with Azure HDInsight clusters topic. The process to upload the .csv files to the default container on the storage account can be found on this page.

How to submit Hive queries

Hive queries can be submitted by using:

- Submit Hive queries through Hadoop Command Line in headnode of Hadoop cluster
- Submit Hive queries with the Hive Editor
- Submit Hive queries with Azure PowerShell Commands

Hive queries are SQL-like. If you are familiar with SQL, you may find the Hive for SQL Users Cheat Sheet useful.

When submitting a Hive query, you can also control the destination of the output from Hive queries, whether it be on the screen or to a local file on the head node or to an Azure blob.

Submit Hive queries through Hadoop Command Line in headnode of Hadoop cluster

If the Hive query is complex, submitting it directly in the head node of the Hadoop cluster typically leads to faster turn around than submitting it with a Hive Editor or Azure PowerShell scripts.

Log in to the head node of the Hadoop cluster, open the Hadoop Command Line on the desktop of the head node, and enter command `cd %hive_home%\bin`.

You have three ways to submit Hive queries in the Hadoop Command Line:

- directly
- using `.hql` files
- with the Hive command console

Submit Hive queries directly in Hadoop Command Line.

You can run command like `hive -e "<your hive query>;` to submit simple Hive queries directly in Hadoop Command Line. Here is an example, where the red box outlines the command that submits the Hive query, and the green box outlines the output from the Hive query.

Submit Hive queries in `.hql` files

When the Hive query is more complicated and has multiple lines, editing queries in command line or Hive command console is not practical. An alternative is to use a text editor in the head node of the Hadoop cluster to save the Hive queries in an `.hql` file in a local directory of the head node. Then the Hive query in the `.hql` file can be submitted by using the `-f` argument as follows:

```
hive -f "<path to the .hql file>"
```

Suppress progress status screen print of Hive queries

By default, after Hive query is submitted in Hadoop Command Line, the progress of the Map/Reduce job is printed out on screen. To suppress the screen print of the Map/Reduce job progress, you can use an argument

`-S` ("S" in upper case) in the command line as follows:

```
hive -S -f "<path to the .hql file>"
hive -S -e "<Hive queries>"
```

Submit Hive queries in Hive command console.

You can also first enter the Hive command console by running command `hive` in Hadoop Command Line, and then submit Hive queries in Hive command console. Here is an example. In this example, the two red boxes highlight the commands used to enter the Hive command console, and the Hive query submitted in Hive command console, respectively. The green box highlights the output from the Hive query.

The previous examples directly output the Hive query results on screen. You can also write the output to a local file on the head node, or to an Azure blob. Then, you can use other tools to further analyze the output of Hive queries.

Output Hive query results to a local file. To output Hive query results to a local directory on the head node, you have to submit the Hive query in the Hadoop Command Line as follows:

```
hive -e "<hive query>" > <local path in the head node>
```

In the following example, the output of Hive query is written into a file `hivequeryoutput.txt` in directory `C:\apps\temp`.

Output Hive query results to an Azure blob

You can also output the Hive query results to an Azure blob, within the default container of the Hadoop cluster. The Hive query for this is as follows:

```
insert overwrite directory wasb:///<directory within the default container> <select clause from ...>
```

In the following example, the output of Hive query is written to a blob directory `queryoutputdir` within the default container of the Hadoop cluster. Here, you only need to provide the directory name, without the blob name. An error is thrown if you provide both directory and blob names, such as `wasb:///queryoutputdir/queryoutput.txt`.

If you open the default container of the Hadoop cluster using Azure Storage Explorer, you can see the output of the Hive query as shown in the following figure. You can apply the filter (highlighted by red box) to only retrieve the blob with specified letters in names.

Submit Hive queries with the Hive Editor

You can also use the Query Console (Hive Editor) by entering a URL of the form *https://<Hadoop cluster name>.azurehdinsight.net/Home/HiveEditor* into a web browser. You must be logged in the see this console and so you need your Hadoop cluster credentials here.

Submit Hive queries with Azure PowerShell Commands

You can also use PowerShell to submit Hive queries. For instructions, see Submit Hive jobs using PowerShell.

Create Hive database and tables

The Hive queries are shared in the GitHub repository and can be downloaded from there.

Here is the Hive query that creates a Hive table.

```
create database if not exists <database name>;
CREATE EXTERNAL TABLE if not exists <database name>.<table name>
(
    field1 string,
    field2 int,
    field3 float,
    field4 double,
    ...,
    fieldN string
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '<field separator>' lines terminated by '<line separator>'
STORED AS TEXTFILE LOCATION '<storage location>' TBLPROPERTIES("skip.header.line.count"="1");
```

Here are the descriptions of the fields that you need to plug in and other configurations:

- **<database name>**: the name of the database that you want to create. If you just want to use the default database, the query "*create database...*" can be omitted.
- **<table name>**: the name of the table that you want to create within the specified database. If you want to use the default database, the table can be directly referred by *<table name>* without <database name>.
- **<field separator>**: the separator that delimits fields in the data file to be uploaded to the Hive table.
- **<line separator>**: the separator that delimits lines in the data file.
- **<storage location>**: the Azure Storage location to save the data of Hive tables. If you do not specify

LOCATION <storage location>, the database and the tables are stored in *hive/warehouse/* directory in the default container of the Hive cluster by default. If you want to specify the storage location, the storage location has to be within the default container for the database and tables. This location has to be referred as location relative to the default container of the cluster in the format of *'wasb:///<directory 1>/'* or *'wasb:///<directory 1>/<directory 2>/'*, etc. After the query is executed, the relative directories are created within the default container.

- **TBLPROPERTIES("skip.header.line.count"="1")**: If the data file has a header line, you have to add this property **at the end** of the *create table* query. Otherwise, the header line is loaded as a record to the table. If the data file does not have a header line, this configuration can be omitted in the query.

Load data to Hive tables

Here is the Hive query that loads data into a Hive table.

```
LOAD DATA INPATH '<path to blob data>' INTO TABLE <database name>.<table name>;
```

- **<path to blob data>**: If the blob file to be uploaded to the Hive table is in the default container of the HDInsight Hadoop cluster, the *<path to blob data>* should be in the format *'wasb://<directory in this container>/<blob file name>'*. The blob file can also be in an additional container of the HDInsight Hadoop cluster. In this case, *<path to blob data>* should be in the format *'wasb://<container name>@<storage account name>.blob.core.windows.net/<blob file name>'*.

> **NOTE**
>
> The blob data to be uploaded to Hive table has to be in the default or additional container of the storage account for the Hadoop cluster. Otherwise, the *LOAD DATA* query fails complaining that it cannot access the data.

Advanced topics: partitioned table and store Hive data in ORC format

If the data is large, partitioning the table is beneficial for queries that only need to scan a few partitions of the table. For instance, it is reasonable to partition the log data of a web site by dates.

In addition to partitioning Hive tables, it is also beneficial to store the Hive data in the Optimized Row Columnar (ORC) format. For more information on ORC formatting, see Using ORC files improves performance when Hive is reading, writing, and processing data.

Partitioned table

Here is the Hive query that creates a partitioned table and loads data into it.

```
CREATE EXTERNAL TABLE IF NOT EXISTS <database name>.<table name>
(field1 string,
...
fieldN string
)
PARTITIONED BY (<partitionfieldname> vartype) ROW FORMAT DELIMITED FIELDS TERMINATED BY '<field separator>'
    lines terminated by '<line separator>' TBLPROPERTIES("skip.header.line.count"="1");
LOAD DATA INPATH '<path to the source file>' INTO TABLE <database name>.<partitioned table name>
    PARTITION (<partitionfieldname>=<partitionfieldvalue>);
```

When querying partitioned tables, it is recommended to add the partition condition in the **beginning** of the `where` clause, which improves the search efficiency.

```
select
    field1, field2, ..., fieldN
from <database name>.<partitioned table name>
where <partitionfieldname>=<partitionfieldvalue> and ...;
```

Store Hive data in ORC format

You cannot directly load data from blob storage into Hive tables that is stored in the ORC format. Here are the steps that the you need to take to load data from Azure blobs to Hive tables stored in ORC format.

Create an external table `STORED AS TEXTFILE` and load data from blob storage to the table.

```
CREATE EXTERNAL TABLE IF NOT EXISTS <database name>.<external textfile table name>
(
    field1 string,
    field2 int,
    ...
    fieldN date
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '<field separator>'
    lines terminated by '<line separator>' STORED AS TEXTFILE
    LOCATION 'wasb:///<directory in Azure blob>' TBLPROPERTIES("skip.header.line.count"="1");

LOAD DATA INPATH '<path to the source file>' INTO TABLE <database name>.<table name>;
```

Create an internal table with the same schema as the external table in step 1, with the same field delimiter, and store the Hive data in the ORC format.

```
CREATE TABLE IF NOT EXISTS <database name>.<ORC table name>
(
    field1 string,
    field2 int,
    ...
    fieldN date
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '<field separator>' STORED AS ORC;
```

Select data from the external table in step 1 and insert into the ORC table

```
INSERT OVERWRITE TABLE <database name>.<ORC table name>
    SELECT * FROM <database name>.<external textfile table name>;
```

> **NOTE**
>
> If the TEXTFILE table *<database name>.<external textfile table name>* has partitions, in STEP 3, the `SELECT * FROM <database name>.<external textfile table name>` command selects the partition variable as a field in the returned data set. Inserting it into the *<database name>.<ORC table name>* fails since *<database name>.<ORC table name>* does not have the partition variable as a field in the table schema. In this case, you need to specifically select the fields to be inserted to *<database name>.<ORC table name>* as follows:

```
INSERT OVERWRITE TABLE <database name>.<ORC table name> PARTITION (<partition variable>=<partition value>)
    SELECT field1, field2, ..., fieldN
    FROM <database name>.<external textfile table name>
    WHERE <partition variable>=<partition value>;
```

It is safe to drop the *<external text file table name>* when using the following query after all data has been

inserted into *<database name>.<ORC table name>*:

```
DROP TABLE IF EXISTS <database name>.<external textfile table name>;
```

After following this procedure, you should have a table with data in the ORC format ready to use.

Build and optimize tables for fast parallel import of data into SQL Server on an Azure VM

11/19/2021 • 5 minutes to read • Edit Online

This article describes how to build partitioned tables for fast parallel bulk importing of data to a SQL Server database. For big data loading/transfer to a SQL database, importing data to the SQL database and subsequent queries can be improved by using *Partitioned Tables and Views*.

Create a new database and a set of filegroups

- Create a new database, if it doesn't exist already.
- Add database filegroups to the database, which holds the partitioned physical files.
- This can be done with CREATE DATABASE if new or ALTER DATABASE if the database exists already.
- Add one or more files (as needed) to each database filegroup.

> **NOTE**
>
> Specify the target filegroup, which holds data for this partition and the physical database file name(s) where the filegroup data is stored.

The following example creates a new database with three filegroups other than the primary and log groups, containing one physical file in each. The database files are created in the default SQL Server Data folder, as configured in the SQL Server instance. For more information about the default file locations, see File Locations for Default and Named Instances of SQL Server.

```
   DECLARE @data_path nvarchar(256);
   SET @data_path = (SELECT SUBSTRING(physical_name, 1, CHARINDEX(N'master.mdf', LOWER(physical_name)) - 1)
      FROM master.sys.master_files
      WHERE database_id = 1 AND file_id = 1);

   EXECUTE ('
      CREATE DATABASE <database_name>
         ON  PRIMARY
        ( NAME = ''Primary'', FILENAME = ''' + @data_path + '<primary_file_name>.mdf'', SIZE = 4096KB ,
FILEGROWTH = 1024KB ),
         FILEGROUP [filegroup_1]
        ( NAME = ''FileGroup1'', FILENAME = ''' + @data_path + '<file_name_1>.ndf'' , SIZE = 4096KB ,
FILEGROWTH = 1024KB ),
         FILEGROUP [filegroup_2]
        ( NAME = ''FileGroup2'', FILENAME = ''' + @data_path + '<file_name_2>.ndf'' , SIZE = 4096KB ,
FILEGROWTH = 1024KB ),
         FILEGROUP [filegroup_3]
        ( NAME = ''FileGroup3'', FILENAME = ''' + @data_path + '<file_name_3>.ndf'' , SIZE = 102400KB ,
FILEGROWTH = 10240KB )
         LOG ON
        ( NAME = ''LogFileGroup'', FILENAME = ''' + @data_path + '<log_file_name>.ldf'' , SIZE = 1024KB ,
FILEGROWTH = 10%)
    ')
```

Create a partitioned table

To create partitioned table(s) according to the data schema, mapped to the database filegroups created in the previous step, you must first create a partition function and scheme. When data is bulk imported to the partitioned table(s), records are distributed among the filegroups according to a partition scheme, as described below.

1. Create a partition function

Create a partition function This function defines the range of values/boundaries to be included in each individual partition table, for example, to limit partitions by month(some_datetime_field) in the year 2013:

```
CREATE PARTITION FUNCTION <DatetimeFieldPFN>(<datetime_field>)
   AS RANGE RIGHT FOR VALUES (
      '20130201', '20130301', '20130401',
      '20130501', '20130601', '20130701', '20130801',
      '20130901', '20131001', '20131101', '20131201' )
```

2. Create a partition scheme

Create a partition scheme. This scheme maps each partition range in the partition function to a physical filegroup, for example:

```
   CREATE PARTITION SCHEME <DatetimeFieldPScheme> AS
     PARTITION <DatetimeFieldPFN> TO (
     <filegroup_1>, <filegroup_2>, <filegroup_3>, <filegroup_4>,
     <filegroup_5>, <filegroup_6>, <filegroup_7>, <filegroup_8>,
     <filegroup_9>, <filegroup_10>, <filegroup_11>, <filegroup_12> )
```

To verify the ranges in effect in each partition according to the function/scheme, run the following query:

```
SELECT psch.name as PartitionScheme,
         prng.value AS PartitionValue,
         prng.boundary_id AS BoundaryID
FROM sys.partition_functions AS pfun
INNER JOIN sys.partition_schemes psch ON pfun.function_id = psch.function_id
INNER JOIN sys.partition_range_values prng ON prng.function_id=pfun.function_id
WHERE pfun.name = <DatetimeFieldPFN>
```

3. Create a partition table

Create partitioned table(s) according to your data schema, and specify the partition scheme and constraint field used to partition the table, for example:

```
CREATE TABLE <table_name> ( [include schema definition here] )
     ON <TablePScheme>(<partition_field>)
```

For more information, see Create Partitioned Tables and Indexes.

Bulk import the data for each individual partition table

- You may use BCP, BULK INSERT, or other methods such as Microsoft Data Migration. The example provided uses the BCP method.
- Alter the database to change transaction logging scheme to BULK_LOGGED to minimize overhead of logging, for example:

  ```
  ALTER DATABASE <database_name> SET RECOVERY BULK_LOGGED
  ```

- To expedite data loading, launch the bulk import operations in parallel. For tips on expediting bulk importing of big data into SQL Server databases, see Load 1 TB in less than 1 hour.

The following PowerShell script is an example of parallel data loading using BCP.

```
# Set database name, input data directory, and output log directory
# This example loads comma-separated input data files
# The example assumes the partitioned data files are named as <base_file_name>_<partition_number>.csv
# Assumes the input data files include a header line. Loading starts at line number 2.

$dbname = "<database_name>"
$indir  = "<path_to_data_files>"
$logdir = "<path_to_log_directory>"

# Select authentication mode
$sqlauth = 0

# For SQL authentication, set the server and user credentials
$sqlusr = "<user@server>"
$server = "<tcp:serverdns>"
$pass   = "<password>"

# Set number of partitions per table - Should match the number of input data files per table
$numofparts = <number_of_partitions>

# Set table name to be loaded, basename of input data files, input format file, and number of partitions
$tbname = "<table_name>"
$basename = "<base_input_data_filename_no_extension>"
$fmtfile = "<full_path_to_format_file>"

# Create log directory if it does not exist
New-Item -ErrorAction Ignore -ItemType directory -Path $logdir

# BCP example using Windows authentication
$ScriptBlock1 = {
   param($dbname, $tbname, $basename, $fmtfile, $indir, $logdir, $num)
   bcp ($dbname + ".." + $tbname) in ($indir + "\" + $basename + "_" + $num + ".csv") -o ($logdir + "\" +
$tbname + "_" + $num + ".txt") -h "TABLOCK" -F 2 -C "RAW" -f ($fmtfile) -T -b 2500 -t "," -r \n
}

# BCP example using SQL authentication
$ScriptBlock2 = {
   param($dbname, $tbname, $basename, $fmtfile, $indir, $logdir, $num, $sqlusr, $server, $pass)
   bcp ($dbname + ".." + $tbname) in ($indir + "\" + $basename + "_" + $num + ".csv") -o ($logdir + "\" +
$tbname + "_" + $num + ".txt") -h "TABLOCK" -F 2 -C "RAW" -f ($fmtfile) -U $sqlusr -S $server -P $pass -b
2500 -t "," -r \n
}

# Background processing of all partitions
for ($i=1; $i -le $numofparts; $i++)
{
   Write-Output "Submit loading trip and fare partitions # $i"
   if ($sqlauth -eq 0) {
      # Use Windows authentication
      Start-Job -ScriptBlock $ScriptBlock1 -Arg ($dbname, $tbname, $basename, $fmtfile, $indir, $logdir, $i)
   }
   else {
      # Use SQL authentication
      Start-Job -ScriptBlock $ScriptBlock2 -Arg ($dbname, $tbname, $basename, $fmtfile, $indir, $logdir, $i,
$sqlusr, $server, $pass)
   }
}

Get-Job

# Optional - Wait till all jobs complete and report date and time
date
While (Get-Job -State "Running") { Start-Sleep 10 }
date
```

Create indexes to optimize joins and query performance

- If you extract data for modeling from multiple tables, create indexes on the join keys to improve the join performance.
- Create indexes (clustered or non-clustered) targeting the same filegroup for each partition, for example:

```
CREATE CLUSTERED INDEX <table_idx> ON <table_name>( [include index columns here] )
    ON <TablePScheme>(<partition)field>)

--  or,

CREATE INDEX <table_idx> ON <table_name>( [include index columns here] )
    ON <TablePScheme>(<partition)field>)
```

> **NOTE**
> You may choose to create the indexes before bulk importing the data. Index creation before bulk importing slows down the data loading.

Advanced Analytics Process and Technology in Action Example

For an end-to-end walkthrough example using the Team Data Science Process with a public dataset, see Team Data Science Process in Action: using SQL Server.

Move data from a SQL Server database to SQL Database with Azure Data Factory

11/19/2021 • 8 minutes to read • Edit Online

This article shows how to move data from a SQL Server database to Azure SQL Database via Azure Blob Storage using the Azure Data Factory (ADF): this method is a supported legacy approach that has the advantages of a replicated staging copy, though we suggest to look at our data migration page for the latest options.

For a table that summarizes various options for moving data to an Azure SQL Database, see Move data to an Azure SQL Database for Azure Machine Learning.

Introduction: What is ADF and when should it be used to migrate data?

Azure Data Factory is a fully managed cloud-based data integration service that orchestrates and automates the movement and transformation of data. The key concept in the ADF model is pipeline. A pipeline is a logical grouping of Activities, each of which defines the actions to perform on the data contained in Datasets. Linked services are used to define the information needed for Data Factory to connect to the data resources.

With ADF, existing data processing services can be composed into data pipelines that are highly available and managed in the cloud. These data pipelines can be scheduled to ingest, prepare, transform, analyze, and publish data, and ADF manages and orchestrates the complex data and processing dependencies. Solutions can be quickly built and deployed in the cloud, connecting a growing number of on-premises and cloud data sources.

Consider using ADF:

- when data needs to be continually migrated in a hybrid scenario that accesses both on-premises and cloud resources
- when the data needs transformations or have business logic added to it when being migrated.

ADF allows for the scheduling and monitoring of jobs using simple JSON scripts that manage the movement of data on a periodic basis. ADF also has other capabilities such as support for complex operations. For more information on ADF, see the documentation at Azure Data Factory (ADF).

The Scenario

We set up an ADF pipeline that composes two data migration activities. Together they move data on a daily basis between a SQL Server database and Azure SQL Database. The two activities are:

- Copy data from a SQL Server database to an Azure Blob Storage account
- Copy data from the Azure Blob Storage account to Azure SQL Database.

> **NOTE**
>
> The steps shown here have been adapted from the more detailed tutorial provided by the ADF team: Copy data from a SQL Server database to Azure Blob storage References to the relevant sections of that topic are provided when appropriate.

Prerequisites

This tutorial assumes you have:

- An **Azure subscription**. If you do not have a subscription, you can sign up for a free trial.
- An **Azure storage account**. You use an Azure storage account for storing the data in this tutorial. If you don't have an Azure storage account, see the Create a storage account article. After you have created the storage account, you need to obtain the account key used to access the storage. See Manage storage account access keys.
- Access to an **Azure SQL Database**. If you must set up an Azure SQL Database, the topic Getting Started with Microsoft Azure SQL Database provides information on how to provision a new instance of an Azure SQL Database.
- Installed and configured **Azure PowerShell** locally. For instructions, see How to install and configure Azure PowerShell.

> **NOTE**
> This procedure uses the Azure portal.

Upload the data to your SQL Server instance

We use the NYC Taxi dataset to demonstrate the migration process. The NYC Taxi dataset is available, as noted in that post, on Azure blob storage NYC Taxi Data. The data has two files, the trip_data.csv file, which contains trip details, and the trip_far.csv file, which contains details of the fare paid for each trip. A sample and description of these files are provided in NYC Taxi Trips Dataset Description.

You can either adapt the procedure provided here to a set of your own data or follow the steps as described by using the NYC Taxi dataset. To upload the NYC Taxi dataset into your SQL Server database, follow the procedure outlined in Bulk Import Data into SQL Server database.

Create an Azure Data Factory

The instructions for creating a new Azure Data Factory and a resource group in the Azure portal are provided Create an Azure Data Factory. Name the new ADF instance *adfdsp* and name the resource group created *adfdsprg*.

Install and configure Azure Data Factory Integration Runtime

The Integration Runtime is a customer-managed data integration infrastructure used by Azure Data Factory to provide data integration capabilities across different network environments. This runtime was formerly called "Data Management Gateway".

To set up, follow the instructions for creating a pipeline

Create linked services to connect to the data resources

A linked service defines the information needed for Azure Data Factory to connect to a data resource. We have three resources in this scenario for which linked services are needed:

1. On-premises SQL Server
2. Azure Blob Storage
3. Azure SQL Database

The step-by-step procedure for creating linked services is provided in Create linked services.

Define and create tables to specify how to access the datasets

Create tables that specify the structure, location, and availability of the datasets with the following script-based procedures. JSON files are used to define the tables. For more information on the structure of these files, see Datasets.

> **NOTE**
> You should execute the `Add-AzureAccount` cmdlet before executing the New-AzureDataFactoryTable cmdlet to confirm that the right Azure subscription is selected for the command execution. For documentation of this cmdlet, see Add-AzureAccount.

The JSON-based definitions in the tables use the following names:

- the **table name** in the SQL Server is *nyctaxi_data*
- the **container name** in the Azure Blob Storage account is *containername*

Three table definitions are needed for this ADF pipeline:

1. SQL on-premises Table
2. Blob Table
3. SQL Azure Table

> **NOTE**
> These procedures use Azure PowerShell to define and create the ADF activities. But these tasks can also be accomplished using the Azure portal. For details, see Create datasets.

SQL on-premises Table

The table definition for the SQL Server is specified in the following JSON file:

```
{
    "name": "OnPremSQLTable",
    "properties":
    {
        "location":
        {
            "type": "OnPremisesSqlServerTableLocation",
            "tableName": "nyctaxi_data",
            "linkedServiceName": "adfonpremsql"
        },
        "availability":
        {
            "frequency": "Day",
            "interval": 1,
            "waitOnExternal":
            {
                "retryInterval": "00:01:00",
                "retryTimeout": "00:10:00",
                "maximumRetry": 3
            }
        }
    }
}
```

The column names were not included here. You can subselect on the column names by including them here (for details check the ADF documentation topic.

Copy the JSON definition of the table into a file called *onpremtabledef.json* file and save it to a known location (here assumed to be *C:\temp\onpremtabledef.json*). Create the table in ADF with the following Azure PowerShell cmdlet:

```
New-AzureDataFactoryTable -ResourceGroupName ADFdsprg -DataFactoryName ADFdsp –File
C:\temp\onpremtabledef.json
```

Blob Table

Definition for the table for the output blob location is in the following (this maps the ingested data from on-premises to Azure blob):

```
{
    "name": "OutputBlobTable",
    "properties":
    {
        "location":
        {
            "type": "AzureBlobLocation",
            "folderPath": "containername",
            "format":
            {
                "type": "TextFormat",
                "columnDelimiter": "\t"
            },
            "linkedServiceName": "adfds"
        },
        "availability":
        {
            "frequency": "Day",
            "interval": 1
        }
    }
}
```

Copy the JSON definition of the table into a file called *bloboutputtabledef.json* file and save it to a known location (here assumed to be *C:\temp\bloboutputtabledef.json*). Create the table in ADF with the following Azure PowerShell cmdlet:

```
New-AzureDataFactoryTable -ResourceGroupName adfdsprg -DataFactoryName adfdsp -File
C:\temp\bloboutputtabledef.json
```

SQL Azure Table

Definition for the table for the SQL Azure output is in the following (this schema maps the data coming from the blob):

```
{
    "name": "OutputSQLAzureTable",
    "properties":
    {
        "structure":
        [
            { "name": "column1", "type": "String"},
            { "name": "column2", "type": "String"}
        ],
        "location":
        {
            "type": "AzureSqlTableLocation",
            "tableName": "your_db_name",
            "linkedServiceName": "adfdssqlazure_linked_servicename"
        },
        "availability":
        {
            "frequency": "Day",
            "interval": 1
        }
    }
}
```

Copy the JSON definition of the table into a file called *AzureSqlTable.json* file and save it to a known location (here assumed to be *C:\temp\AzureSqlTable.json*). Create the table in ADF with the following Azure PowerShell cmdlet:

```
New-AzureDataFactoryTable -ResourceGroupName adfdsprg -DataFactoryName adfdsp -File
C:\temp\AzureSqlTable.json
```

Define and create the pipeline

Specify the activities that belong to the pipeline and create the pipeline with the following script-based procedures. A JSON file is used to define the pipeline properties.

- The script assumes that the **pipeline name** is *AMLDSProcessPipeline*.
- Also note that we set the periodicity of the pipeline to be executed on daily basis and use the default execution time for the job (12 am UTC).

> **NOTE**
> The following procedures use Azure PowerShell to define and create the ADF pipeline. But this task can also be accomplished using the Azure portal. For details, see Create pipeline.

Using the table definitions provided previously, the pipeline definition for the ADF is specified as follows:

```
{
    "name": "AMLDSProcessPipeline",
    "properties":
    {
        "description" : "This pipeline has two activities: the first one copies data from SQL Server to
Azure Blob, and the second one copies from Azure Blob to Azure Database Table",
        "activities":
        [
            {
                "name": "CopyFromSQLtoBlob",
                "description": "Copy data from SQL Server to blob",
                "type": "CopyActivity",
                "inputs": [ {"name": "OnPremSQLTable"} ],
                "outputs": [ {"name": "OutputBlobTable"} ],
                "transformation":
                {
                    "source":
                    {
                        "type": "SqlSource",
                        "sqlReaderQuery": "select * from nyctaxi_data"
                    },
                    "sink":
                    {
                        "type": "BlobSink"
                    }
                },
                "Policy":
                {
                    "concurrency": 3,
                    "executionPriorityOrder": "NewestFirst",
                    "style": "StartOfInterval",
                    "retry": 0,
                    "timeout": "01:00:00"
                }
            },
            {
                "name": "CopyFromBlobtoSQLAzure",
                "description": "Push data to Sql Azure",
                "type": "CopyActivity",
                "inputs": [ {"name": "OutputBlobTable"} ],
                "outputs": [ {"name": "OutputSQLAzureTable"} ],
                "transformation":
                {
                    "source":
                    {
                        "type": "BlobSource"
                    },
                    "sink":
                    {
                        "type": "SqlSink",
                        "WriteBatchTimeout": "00:5:00",
                    }
                },
                "Policy":
                {
                    "concurrency": 3,
                    "executionPriorityOrder": "NewestFirst",
                    "style": "StartOfInterval",
                    "retry": 2,
                    "timeout": "02:00:00"
                }
            }
        ]
    }
}
```

Copy this JSON definition of the pipeline into a file called *pipelinedef.json* file and save it to a known location (here assumed to be *C:\temp\pipelinedef.json*). Create the pipeline in ADF with the following Azure PowerShell cmdlet:

```
New-AzureDataFactoryPipeline  -ResourceGroupName adfdsprg -DataFactoryName adfdsp -File
C:\temp\pipelinedef.json
```

Start the Pipeline

The pipeline can now be run using the following command:

```
Set-AzureDataFactoryPipelineActivePeriod -ResourceGroupName ADFdsprg -DataFactoryName ADFdsp -StartDateTime
startdateZ –EndDateTime enddateZ –Name AMLDSProcessPipeline
```

The *startdate* and *enddate* parameter values need to be replaced with the actual dates between which you want the pipeline to run.

Once the pipeline executes, you should be able to see the data show up in the container selected for the blob, one file per day.

We have not leveraged the functionality provided by ADF to pipe data incrementally. For more information on how to do this and other capabilities provided by ADF, see the ADF documentation.

Prepare data for enhanced machine learning

11/19/2021 • 6 minutes to read • Edit Online

Pre-processing and cleaning data are important tasks that must be conducted before a dataset can be used for model training. Raw data is often noisy and unreliable, and may be missing values. Using such data for modeling can produce misleading results. These tasks are part of the Team Data Science Process (TDSP) and typically follow an initial exploration of a dataset used to discover and plan the pre-processing required. For more detailed instructions on the TDSP process, see the steps outlined in the Team Data Science Process.

Pre-processing and cleaning tasks, like the data exploration task, can be carried out in a wide variety of environments, such as SQL or Hive or Azure Machine Learning Studio (classic), and with various tools and languages, such as R or Python, depending where your data is stored and how it is formatted. Since TDSP is iterative in nature, these tasks can take place at various steps in the workflow of the process.

This article introduces various data processing concepts and tasks that can be undertaken either before or after ingesting data into Azure Machine Learning Studio (classic).

For an example of data exploration and pre-processing done inside Azure Machine Learning Studio (classic), see the Pre-processing data video.

Why pre-process and clean data?

Real world data is gathered from various sources and processes and it may contain irregularities or corrupt data compromising the quality of the dataset. The typical data quality issues that arise are:

- **Incomplete**: Data lacks attributes or containing missing values.
- **Noisy**: Data contains erroneous records or outliers.
- **Inconsistent**: Data contains conflicting records or discrepancies.

Quality data is a prerequisite for quality predictive models. To avoid "garbage in, garbage out" and improve data quality and therefore model performance, it is imperative to conduct a data health screen to spot data issues early and decide on the corresponding data processing and cleaning steps.

What are some typical data health screens that are employed?

We can check the general quality of data by checking:

- The number of **records**.
- The number of **attributes** (or **features**).
- The attribute **data types** (nominal, ordinal, or continuous).
- The number of **missing values**.
- **Well-formed** data.
  - If the data is in TSV or CSV, check that the column separators and line separators always correctly separate columns and lines.
  - If the data is in HTML or XML format, check whether the data is well formed based on their respective standards.
  - Parsing may also be necessary in order to extract structured information from semi-structured or unstructured data.
- **Inconsistent data records**. Check the range of values are allowed. For example, if the data contains student GPA (grade point average), check if the GPA is in the designated range, say 0~4.

When you find issues with data, **processing steps** are necessary, which often involves cleaning missing values, data normalization, discretization, text processing to remove and/or replace embedded characters that may affect data alignment, mixed data types in common fields, and others.

Azure Machine Learning consumes well-formed tabular data. If the data is already in tabular form, data pre-processing can be performed directly with Azure Machine Learning Studio (classic) in the Machine Learning. If data is not in tabular form, say it is in XML, parsing may be required in order to convert the data to tabular form.

What are some of the major tasks in data pre-processing?

- **Data cleaning**: Fill in missing values, detect, and remove noisy data and outliers.
- **Data transformation**: Normalize data to reduce dimensions and noise.
- **Data reduction**: Sample data records or attributes for easier data handling.
- **Data discretization**: Convert continuous attributes to categorical attributes for ease of use with certain machine learning methods.
- **Text cleaning**: remove embedded characters that may cause data misalignment, for example, embedded tabs in a tab-separated data file, embedded new lines that may break records, for example.

The sections below detail some of these data processing steps.

How to deal with missing values?

To deal with missing values, it is best to first identify the reason for the missing values to better handle the problem. Typical missing value handling methods are:

- **Deletion**: Remove records with missing values
- **Dummy substitution**: Replace missing values with a dummy value: e.g, *unknown* for categorical or 0 for numerical values.
- **Mean substitution**: If the missing data is numerical, replace the missing values with the mean.
- **Frequent substitution**: If the missing data is categorical, replace the missing values with the most frequent item
- **Regression substitution**: Use a regression method to replace missing values with regressed values.

How to normalize data?

Data normalization rescales numerical values to a specified range. Popular data normalization methods include:

- **Min-Max Normalization**: Linearly transform the data to a range, say between 0 and 1, where the min value is scaled to 0 and max value to 1.
- **Z-score Normalization**: Scale data based on mean and standard deviation: divide the difference between the data and the mean by the standard deviation.
- **Decimal scaling**: Scale the data by moving the decimal point of the attribute value.

How to discretize data?

Data can be discretized by converting continuous values to nominal attributes or intervals. Some ways of doing this are:

- **Equal-Width Binning**: Divide the range of all possible values of an attribute into N groups of the same size, and assign the values that fall in a bin with the bin number.
- **Equal-Height Binning**: Divide the range of all possible values of an attribute into N groups, each containing the same number of instances, then assign the values that fall in a bin with the bin number.

How to reduce data?

There are various methods to reduce data size for easier data handling. Depending on data size and the domain, the following methods can be applied:

- **Record Sampling**: Sample the data records and only choose the representative subset from the data.
- **Attribute Sampling**: Select only a subset of the most important attributes from the data.
- **Aggregation**: Divide the data into groups and store the numbers for each group. For example, the daily revenue numbers of a restaurant chain over the past 20 years can be aggregated to monthly revenue to reduce the size of the data.

How to clean text data?

Text fields in tabular data may include characters that affect columns alignment and/or record boundaries. For example, embedded tabs in a tab-separated file cause column misalignment, and embedded new line characters break record lines. Improper text encoding handling while writing or reading text leads to information loss, inadvertent introduction of unreadable characters (like nulls), and may also affect text parsing. Careful parsing and editing may be required in order to clean text fields for proper alignment and/or to extract structured data from unstructured or semi-structured text data.

Data exploration offers an early view into the data. A number of data issues can be uncovered during this step and corresponding methods can be applied to address those issues. It is important to ask questions such as what is the source of the issue and how the issue may have been introduced. This process also helps you decide on the data processing steps that need to be taken to resolve them. Identifying the final use cases and personas can also be used to prioritize the data processing effort.

References

> *Data Mining: Concepts and Techniques*, Third Edition, Morgan Kaufmann, 2011, Jiawei Han, Micheline Kamber, and Jian Pei

Explore data in the Team Data Science Process

11/19/2021 • 2 minutes to read • Edit Online

Exploring data is a step in the Team Data Science Process.

The following articles describe how to explore data in three different storage environments that are typically used in the Data Science Process:

- Explore Azure blob container data using the Pandas Python package.
- Explore SQL Server data by using SQL and by using a programming language like Python.
- Explore Hive table data using Hive queries.

In addition, the video, Preprocessing Data in Azure Machine Learning Studio, describes the commonly used modules for cleaning and transforming data.

Explore data in Azure Blob storage with the pandas Python package

11/19/2021 • 2 minutes to read • Edit Online

This article covers how to explore data that is stored in Azure blob container using the pandas Python package.

This task is a step in the Team Data Science Process.

Prerequisites

This article assumes that you have:

- Created an Azure storage account. If you need instructions, see Create an Azure Storage account
- Stored your data in an Azure Blob storage account. If you need instructions, see Moving data to and from Azure Storage

Load the data into a pandas DataFrame

To explore and manipulate a dataset, it must first be downloaded from the blob source to a local file, which can then be loaded in a pandas DataFrame. Here are the steps to follow for this procedure:

1. Download the data from Azure blob with the following Python code sample using Blob service. Replace the variable in the following code with your specific values:

```
from azure.storage.blob import BlobServiceClient
import pandas as pd

STORAGEACCOUNTURL= <storage_account_url>
STORAGEACCOUNTKEY= <storage_account_key>
LOCALFILENAME= <local_file_name>
CONTAINERNAME= <container_name>
BLOBNAME= <blob_name>

#download from blob
t1=time.time()
blob_service_client_instance = BlobServiceClient(account_url=STORAGEACCOUNTURL,
credential=STORAGEACCOUNTKEY)
blob_client_instance = blob_service_client_instance.get_blob_client(CONTAINERNAME, BLOBNAME,
snapshot=None)
with open(LOCALFILENAME, "wb") as my_blob:
    blob_data = blob_client_instance.download_blob()
    blob_data.readinto(my_blob)
t2=time.time()
print(("It takes %s seconds to download "+BLOBNAME) % (t2 - t1))
```

2. Read the data into a pandas DataFrame from the downloaded file.

```
# LOCALFILE is the file path
dataframe_blobdata = pd.read_csv(LOCALFILENAME)
```

If you need more general information on reading from an Azure Storage Blob, look at our documentation Azure Storage Blobs client library for Python.

Now you are ready to explore the data and generate features on this dataset.

Examples of data exploration using pandas

Here are a few examples of ways to explore data using pandas:

1. Inspect the **number of rows and columns**

```
print('the size of the data is: %d rows and  %d columns' % dataframe_blobdata.shape)
```

2. **Inspect** the first or last few **rows** in the following dataset:

```
dataframe_blobdata.head(10)

dataframe_blobdata.tail(10)
```

3. Check the **data type** each column was imported as using the following sample code

```
for col in dataframe_blobdata.columns:
    print(dataframe_blobdata[col].name, ':\t', dataframe_blobdata[col].dtype)
```

4. Check the **basic stats** for the columns in the data set as follows

```
dataframe_blobdata.describe()
```

5. Look at the number of entries for each column value as follows

```
dataframe_blobdata['<column_name>'].value_counts()
```

6. **Count missing values** versus the actual number of entries in each column using the following sample code

```
miss_num = dataframe_blobdata.shape[0] - dataframe_blobdata.count()
print(miss_num)
```

7. If you have **missing values** for a specific column in the data, you can drop them as follows:

```
dataframe_blobdata_noNA = dataframe_blobdata.dropna()
dataframe_blobdata_noNA.shape
```

   Another way to replace missing values is with the mode function:

```
dataframe_blobdata_mode = dataframe_blobdata.fillna(
    {'<column_name>': dataframe_blobdata['<column_name>'].mode()[0]})
```

8. Create a **histogram** plot using variable number of bins to plot the distribution of a variable

```
dataframe_blobdata['<column_name>'].value_counts().plot(kind='bar')

np.log(dataframe_blobdata['<column_name>']+1).hist(bins=50)
```

9. Look at **correlations** between variables using a scatterplot or using the built-in correlation function

```
# relationship between column_a and column_b using scatter plot
plt.scatter(dataframe_blobdata['<column_a>'], dataframe_blobdata['<column_b>'])

# correlation between column_a and column_b
dataframe_blobdata[['<column_a>', '<column_b>']].corr()
```

Explore data in a SQL Server virtual machine on Azure

11/19/2021 • 2 minutes to read • Edit Online

This article covers how to explore data that is stored in a SQL Server VM on Azure. Use SQL or Python to examine the data.

This task is a step in the Team Data Science Process.

> **NOTE**
> The sample SQL statements in this document assume that data is in SQL Server. If it isn't, refer to the cloud data science process map to learn how to move your data to SQL Server.

Explore SQL data with SQL scripts

Here are a few sample SQL scripts that can be used to explore data stores in SQL Server.

1. Get the count of observations per day

```
SELECT CONVERT(date, <date_columnname>) as date, count(*) as c from <tablename> group by CONVERT(date,
<date_columnname>)
```

2. Get the levels in a categorical column

```
select distinct <column_name> from <databasename>
```

3. Get the number of levels in combination of two categorical columns

```
select <column_a>, <column_b>,count(*) from <tablename> group by <column_a>, <column_b>
```

4. Get the distribution for numerical columns

```
select <column_name>, count(*) from <tablename> group by <column_name>
```

> **NOTE**
> For a practical example, you can use the NYC Taxi dataset and refer to the IPNB titled NYC Data wrangling using IPython Notebook and SQL Server for an end-to-end walk-through.

Explore SQL data with Python

Using Python to explore data and generate features when the data is in SQL Server is similar to processing data in Azure blob using Python, as documented in Process Azure Blob data in your data science environment. Load the data from the database into a pandas DataFrame and then can be processed further. We document the process of connecting to the database and loading the data into the DataFrame in this section.

The following connection string format can be used to connect to a SQL Server database from Python using pyodbc (replace servername, dbname, username, and password with your specific values):

```
#Set up the SQL Azure connection
import pyodbc
conn = pyodbc.connect('DRIVER={SQL Server};SERVER=<servername>;DATABASE=<dbname>;UID=<username>;PWD=
<password>')
```

The Pandas library in Python provides a rich set of data structures and data analysis tools for data manipulation for Python programming. The following code reads the results returned from a SQL Server database into a Pandas data frame:

```
# Query database and load the returned results in pandas data frame
data_frame = pd.read_sql('''select <columnname1>, <columnname2>... from <tablename>''', conn)
```

Now you can work with the Pandas DataFrame as covered in the topic Process Azure Blob data in your data science environment.

The Team Data Science Process in action example

For an end-to-end walkthrough example of the Cortana Analytics Process using a public dataset, see The Team Data Science Process in action: using SQL Server.

Explore data in Hive tables with Hive queries

11/19/2021 • 2 minutes to read • Edit Online

This article provides sample Hive scripts that are used to explore data in Hive tables in an HDInsight Hadoop cluster.

This task is a step in the Team Data Science Process.

Prerequisites

This article assumes that you have:

- Created an Azure storage account. If you need instructions, see Create an Azure Storage account
- Provisioned a customized Hadoop cluster with the HDInsight service. If you need instructions, see Customize Azure HDInsight Hadoop Clusters for Advanced Analytics.
- The data has been uploaded to Hive tables in Azure HDInsight Hadoop clusters. If it has not, follow the instructions in Create and load data to Hive tables to upload data to Hive tables first.
- Enabled remote access to the cluster. If you need instructions, see Access the Head Node of Hadoop Cluster.
- If you need instructions on how to submit Hive queries, see How to Submit Hive Queries

Example Hive query scripts for data exploration

1. Get the count of observations per partition

```
SELECT <partitionfieldname>, count(*) from <databasename>.<tablename> group by <partitionfieldname>;
```

2. Get the count of observations per day

```
SELECT to_date(<date_columnname>), count(*) from <databasename>.<tablename> group by
to_date(<date_columnname>);
```

3. Get the levels in a categorical column

```
SELECT distinct <column_name> from <databasename>.<tablename>
```

4. Get the number of levels in combination of two categorical columns

```
SELECT <column_a>, <column_b>, count(*) from <databasename>.<tablename> group by <column_a>,
<column_b>
```

5. Get the distribution for numerical columns

```
SELECT <column_name>, count(*) from <databasename>.<tablename> group by <column_name>
```

6. Extract records from joining two tables

```
SELECT
    a.<common_columnname1> as <new_name1>,
    a.<common_columnname2> as <new_name2>,
    a.<a_column_name1> as <new_name3>,
    a.<a_column_name2> as <new_name4>,
    b.<b_column_name1> as <new_name5>,
    b.<b_column_name2> as <new_name6>
FROM
    (
    SELECT <common_columnname1>,
        <common_columnname2>,
        <a_column_name1>,
        <a_column_name2>,
    FROM <databasename>.<tablename1>
    ) a
    join
    (
    SELECT <common_columnname1>,
        <common_columnname2>,
        <b_column_name1>,
        <b_column_name2>,
    FROM <databasename>.<tablename2>
    ) b
    ON a.<common_columnname1>=b.<common_columnname1> and a.<common_columnname2>=b.
<common_columnname2>
```

Additional query scripts for taxi trip data scenarios

Examples of queries that are specific to NYC Taxi Trip Data scenarios are also provided in GitHub repository. These queries already have data schema specified and are ready to be submitted to run.

Sample data in Azure blob containers, SQL Server, and Hive tables

11/19/2021 • 2 minutes to read • Edit Online

The following articles describe how to sample data that is stored in one of three different Azure locations:

- **Azure blob container data** is sampled by downloading it programmatically and then sampling it with sample Python code.
- **SQL Server data** is sampled using both SQL and the Python Programming Language.
- **Hive table data** is sampled using Hive queries.

This sampling task is a step in the Team Data Science Process (TDSP).

Why sample data?

If the dataset you plan to analyze is large, it's usually a good idea to down-sample the data to reduce it to a smaller but representative and more manageable size. Downsizing may facilitate data understanding, exploration, and feature engineering. This sampling role in the Cortana Analytics Process is to enable fast prototyping of the data processing functions and machine learning models.

Sample data in Azure Blob storage

11/19/2021 • 2 minutes to read • Edit Online

This article covers sampling data stored in Azure Blob storage by downloading it programmatically and then sampling it using procedures written in Python.

Why sample your data? If the dataset you plan to analyze is large, it's usually a good idea to down-sample the data to reduce it to a smaller but representative and more manageable size. Sampling facilitates data understanding, exploration, and feature engineering. Its role in the Cortana Analytics Process is to enable fast prototyping of the data processing functions and machine learning models.

This sampling task is a step in the Team Data Science Process (TDSP).

Download and down-sample data

1. Download the data from Azure Blob storage using the Blob service from the following sample Python code:

```
from azure.storage.blob import BlobService
import tables

STORAGEACCOUNTNAME= <storage_account_name>
STORAGEACCOUNTKEY= <storage_account_key>
LOCALFILENAME= <local_file_name>
CONTAINERNAME= <container_name>
BLOBNAME= <blob_name>

#download from blob
t1=time.time()
blob_service=BlobService(account_name=STORAGEACCOUNTNAME,account_key=STORAGEACCOUNTKEY)
blob_service.get_blob_to_path(CONTAINERNAME,BLOBNAME,LOCALFILENAME)
t2=time.time()
print(("It takes %s seconds to download "+blobname) % (t2 - t1))
```

2. Read data into a Pandas data-frame from the file downloaded above.

```
import pandas as pd

#directly ready from file on disk
dataframe_blobdata = pd.read_csv(LOCALFILE)
```

3. Down-sample the data using the `numpy`'s `random.choice` as follows:

```
# A 1 percent sample
sample_ratio = 0.01
sample_size = np.round(dataframe_blobdata.shape[0] * sample_ratio)
sample_rows = np.random.choice(dataframe_blobdata.index.values, sample_size)
dataframe_blobdata_sample = dataframe_blobdata.ix[sample_rows]
```

Now you can work with the above data frame with the one Percent sample for further exploration and feature generation.

Upload data and read it into Azure Machine Learning

You can use the following sample code to down-sample the data and use it directly in Azure Machine Learning:

1. Write the data frame to a local file

```
dataframe.to_csv(os.path.join(os.getcwd(),LOCALFILENAME), sep='\t', encoding='utf-8', index=False)
```

2. Upload the local file to an Azure blob using the following sample code:

```
from azure.storage.blob import BlobService
import tables

STORAGEACCOUNTNAME= <storage_account_name>
LOCALFILENAME= <local_file_name>
STORAGEACCOUNTKEY= <storage_account_key>
CONTAINERNAME= <container_name>
BLOBNAME= <blob_name>

output_blob_service=BlobService(account_name=STORAGEACCOUNTNAME,account_key=STORAGEACCOUNTKEY)
localfileprocessed = os.path.join(os.getcwd(),LOCALFILENAME) #assuming file is in current working
directory

try:

#perform upload
output_blob_service.put_block_blob_from_path(CONTAINERNAME,BLOBNAME,localfileprocessed)

except:
    print ("Something went wrong with uploading to the blob:"+ BLOBNAME)
```

3. Read the data from the Azure blob using Azure Machine Learning Import Data as shown in the image below:

Sample data in SQL Server on Azure

11/19/2021 • 3 minutes to read • Edit Online

This article shows how to sample data stored in SQL Server on Azure using either SQL or the Python programming language. It also shows how to move sampled data into Azure Machine Learning by saving it to a file, uploading it to an Azure blob, and then reading it into Azure Machine Learning Studio.

The Python sampling uses the pyodbc ODBC library to connect to SQL Server on Azure and the Pandas library to do the sampling.

> **NOTE**
> The sample SQL code in this document assumes that the data is in a SQL Server on Azure. If it is not, refer to Move data to SQL Server on Azure article for instructions on how to move your data to SQL Server on Azure.

Why sample your data? If the dataset you plan to analyze is large, it's usually a good idea to down-sample the data to reduce it to a smaller but representative and more manageable size. Sampling facilitates data understanding, exploration, and feature engineering. Its role in the Team Data Science Process (TDSP) is to enable fast prototyping of the data processing functions and machine learning models.

This sampling task is a step in the Team Data Science Process (TDSP).

Using SQL

This section describes several methods using SQL to perform simple random sampling against the data in the database. Choose a method based on your data size and its distribution.

The following two items show how to use `newid` in SQL Server to perform the sampling. The method you choose depends on how random you want the sample to be (pk_id in the following sample code is assumed to be an autogenerated primary key).

1. Less strict random sample

```
select  * from <table_name> where <primary_key> in
(select top 10 percent <primary_key> from <table_name> order by newid())
```

2. More random sample

```
SELECT * FROM <table_name>
WHERE 0.1 >= CAST(CHECKSUM(NEWID(), <primary_key>) & 0x7fffffff AS float)/ CAST (0x7fffffff AS int)
```

Tablesample can be used for sampling the data as well. This option may be a better approach if your data size is large (assuming that data on different pages is not correlated) and for the query to complete in a reasonable time.

```
SELECT *
FROM <table_name>
TABLESAMPLE (10 PERCENT)
```

> **NOTE**
> You can explore and generate features from this sampled data by storing it in a new table

Connecting to Azure Machine Learning

You can directly use the sample queries above in the Azure Machine Learning Import Data module to down-sample the data on the fly and bring it into an Azure Machine Learning experiment. A screenshot of using the reader module to read the sampled data is shown here:

Using the Python programming language

This section demonstrates using the pyodbc library to establish an ODBC connect to a SQL server database in Python. The database connection string is as follows: (replace servername, dbname, username, and password with your configuration):

```
#Set up the SQL Azure connection
import pyodbc
conn = pyodbc.connect('DRIVER={SQL Server};SERVER=<servername>;DATABASE=<dbname>;UID=<username>;PWD=
<password>')
```

The Pandas library in Python provides a rich set of data structures and data analysis tools for data manipulation for Python programming. The following code reads a 0.1% sample of the data from a table in Azure SQL Database into a Pandas data:

```
import pandas as pd

# Query database and load the returned results in pandas data frame
data_frame = pd.read_sql('''select column1, column2... from <table_name> tablesample (0.1 percent)''', conn)
```

You can now work with the sampled data in the Pandas data frame.

Connecting to Azure Machine Learning

You can use the following sample code to save the down-sampled data to a file and upload it to an Azure blob. The data in the blob can be directly read into an Azure Machine Learning Experiment using the Import Data module. The steps are as follows:

1. Write the pandas data frame to a local file

```
dataframe.to_csv(os.path.join(os.getcwd(),LOCALFILENAME), sep='\t', encoding='utf-8', index=False)
```

2. Upload local file to Azure blob

```
from azure.storage import BlobService
import tables

STORAGEACCOUNTNAME= <storage_account_name>
LOCALFILENAME= <local_file_name>
STORAGEACCOUNTKEY= <storage_account_key>
CONTAINERNAME= <container_name>
BLOBNAME= <blob_name>

output_blob_service=BlobService(account_name=STORAGEACCOUNTNAME,account_key=STORAGEACCOUNTKEY)
localfileprocessed = os.path.join(os.getcwd(),LOCALFILENAME) #assuming file is in current working
directory

try:

#perform upload
output_blob_service.put_block_blob_from_path(CONTAINERNAME,BLOBNAME,localfileprocessed)

except:
    print ("Something went wrong with uploading blob:"+BLOBNAME)
```

3. Read data from Azure blob using Azure Machine Learning Import Data module as shown in the following screen grab:

The Team Data Science Process in Action example

To walk through an example of the Team Data Science Process a using a public dataset, see Team Data Science Process in Action: using SQL Server.

Sample data in Azure HDInsight Hive tables

11/19/2021 • 2 minutes to read • Edit Online

This article describes how to down-sample data stored in Azure HDInsight Hive tables using Hive queries to reduce it to a size more manageable for analysis. It covers three popularly used sampling methods:

- Uniform random sampling
- Random sampling by groups
- Stratified sampling

Why sample your data? If the dataset you plan to analyze is large, it's usually a good idea to down-sample the data to reduce it to a smaller but representative and more manageable size. Down-sampling facilitates data understanding, exploration, and feature engineering. Its role in the Team Data Science Process is to enable fast prototyping of the data processing functions and machine learning models.

This sampling task is a step in the Team Data Science Process (TDSP).

How to submit Hive queries

Hive queries can be submitted from the Hadoop Command-Line console on the head node of the Hadoop cluster. Log into the head node of the Hadoop cluster, open the Hadoop Command-Line console, and submit the Hive queries from there. For instructions on submitting Hive queries in the Hadoop Command-Line console, see How to Submit Hive Queries.

Uniform random sampling

Uniform random sampling means that each row in the data set has an equal chance of being sampled. It can be implemented by adding an extra field rand() to the data set in the inner "select" query, and in the outer "select" query that condition on that random field.

Here is an example query:

```
SET sampleRate=<sample rate, 0-1>;
select
    field1, field2, …, fieldN
from
    (
    select
        field1, field2, …, fieldN, rand() as samplekey
    from <hive table name>
    )a
where samplekey<='${hiveconf:sampleRate}'
```

Here, `<sample rate, 0-1>` specifies the proportion of records that the users want to sample.

Random sampling by groups

When sampling categorical data, you may want to either include or exclude all of the instances for some value of the categorical variable. This sort of sampling is called "sampling by group". For example, if you have a categorical variable "*State*", which has values such as NY, MA, CA, NJ, and PA, you want records from each state to be together, whether they are sampled or not.

Here is an example query that samples by group:

```
SET sampleRate=<sample rate, 0-1>;
select
    b.field1, b.field2, …, b.catfield, …, b.fieldN
from
    (
    select
        field1, field2, …, catfield, …, fieldN
    from <table name>
    )b
join
    (
    select
        catfield
    from
        (
        select
            catfield, rand() as samplekey
        from <table name>
        group by catfield
        )a
    where samplekey<='${hiveconf:sampleRate}'
    )c
on b.catfield=c.catfield
```

Stratified sampling

Random sampling is stratified with respect to a categorical variable when the samples obtained have categorical values that are present in the same ratio as they were in the parent population. Using the same example as above, suppose your data has the following observations by states: NJ has 100 observations, NY has 60 observations, and WA has 300 observations. If you specify the rate of stratified sampling to be 0.5, then the sample obtained should have approximately 50, 30, and 150 observations of NJ, NY, and WA respectively.

Here is an example query:

```
SET sampleRate=<sample rate, 0-1>;
select
    field1, field2, field3, ..., fieldN, state
from
    (
    select
        field1, field2, field3, ..., fieldN, state,
        count(*) over (partition by state) as state_cnt,
          rank() over (partition by state order by rand()) as state_rank
      from <table name>
    ) a
where state_rank <= state_cnt*'${hiveconf:sampleRate}'
```

For information on more advanced sampling methods that are available in Hive, see LanguageManual Sampling.

Access datasets with Python using the Azure Machine Learning Python client library

11/19/2021 • 8 minutes to read • Edit Online

The preview of Microsoft Azure Machine Learning Python client library can enable secure access to your Azure Machine Learning datasets from a local Python environment and enables the creation and management of datasets in a workspace.

This topic provides instructions on how to:

- install the Machine Learning Python client library
- access and upload datasets, including instructions on how to get authorization to access Azure Machine Learning datasets from your local Python environment
- access intermediate datasets from experiments
- use the Python client library to enumerate datasets, access metadata, read the contents of a dataset, create new datasets, and update existing datasets

Prerequisites

The Python client library has been tested under the following environments:

- Windows, Mac, and Linux
- Python 2.7 and 3.6+

It has a dependency on the following packages:

- requests
- python-dateutil
- pandas

We recommend using a Python distribution such as Anaconda or Canopy, which come with Python, IPython and the three packages listed above installed. Although IPython is not strictly required, it is a great environment for manipulating and visualizing data interactively.

How to install the Azure Machine Learning Python client library

Install the Azure Machine Learning Python client library to complete the tasks outlined in this topic. This library is available from the Python Package Index. To install it in your Python environment, run the following command from your local Python environment:

```
pip install azureml
```

Alternatively, you can download and install from the sources on GitHub.

```
python setup.py install
```

If you have git installed on your machine, you can use pip to install directly from the git repository:

```
pip install git+https://github.com/Azure/Azure-MachineLearning-ClientLibrary-Python.git
```

Use code snippets to access datasets

The Python client library gives you programmatic access to your existing datasets from experiments that have been run.

From the Azure Machine Learning Studio (classic) web interface, you can generate code snippets that include all the necessary information to download and deserialize datasets as pandas DataFrame objects on your local machine.

Security for data access

The code snippets provided by Azure Machine Learning Studio (classic) for use with the Python client library includes your workspace ID and authorization token. These provide full access to your workspace and must be protected, like a password.

For security reasons, the code snippet functionality is only available to users that have their role set as **Owner** for the workspace. Your role is displayed in Azure Machine Learning Studio (classic) on the **USERS** page under **Settings**.

If your role is not set as **Owner**, you can either request to be reinvited as an owner, or ask the owner of the workspace to provide you with the code snippet.

To obtain the authorization token, you may choose one of these options:

- Ask for a token from an owner. Owners can access their authorization tokens from the Settings page of their workspace in Azure Machine Learning Studio (classic). Select **Settings** from the left pane and click **AUTHORIZATION TOKENS** to see the primary and secondary tokens. Although either the primary or the secondary authorization tokens can be used in the code snippet, it is recommended that owners only share the secondary authorization tokens.

- Ask to be promoted to role of owner: a current owner of the workspace needs to first remove you from the workspace then reinvite you to it as an owner.

Once developers have obtained the workspace ID and authorization token, they are able to access the workspace using the code snippet regardless of their role.

Authorization tokens are managed on the **AUTHORIZATION TOKENS** page under **SETTINGS**. You can regenerate them, but this procedure revokes access to the previous token.

Access datasets from a local Python application

1. In Machine Learning Studio (classic), click **DATASETS** in the navigation bar on the left.

2. Select the dataset you would like to access. You can select any of the datasets from the **MY DATASETS** list or from the **SAMPLES** list.

3. From the bottom toolbar, click **Generate Data Access Code**. If the data is in a format incompatible with the Python client library, this button is disabled.

4. Select the code snippet from the window that appears and copy it to your clipboard.

5. Paste the code into the notebook of your local Python application.

	fLength	fWidth	fSize	fConc	fConcl	fAsym	fM3Long	fM3Trans	fAlpha	fDist	Class
0	29.4491	12.7271	2.6637	0.3536	0.1876	-19.4070	-18.1295	7.1258	8.1501	252.1500	g
1	51.5830	10.7969	2.6222	0.5227	0.2733	-64.0583	-30.1280	4.3855	15.0428	269.4810	g
2	36.3558	10.3843	2.8531	0.5309	0.3599	15.4179	51.9328	16.2848	9.1263	208.7935	h
3	37.2577	12.0793	2.4354	0.3560	0.2037	5.1882	-17.8545	6.0370	30.0150	61.1727	h
4	34.8906	15.7072	2.7147	0.3587	0.1938	-8.5682	-12.6514	-14.3676	89.7920	222.4690	h
5	60.4957	17.6753	2.9380	0.1753	0.0894	31.3257	-15.9801	14.4117	15.6390	113.1820	g
6	18.5731	16.2365	2.6758	0.4895	0.3091	1.1158	8.1104	-11.7988	50.9791	224.8780	h
7	21.5929	12.4539	2.3512	0.4900	0.3096	0.1172	-4.3583	2.5525	58.3290	52.0772	g
8	45.5590	9.9957	2.5809	0.3885	0.1955	17.0284	34.9608	-7.8856	14.4173	177.6820	g
9	62.3597	21.6946	3.1739	0.3087	0.2041	-45.3412	52.1023	22.5905	36.9876	274.7409	h

Access intermediate datasets from Machine Learning experiments

After an experiment is run in Machine Learning Studio (classic), it is possible to access the intermediate datasets from the output nodes of modules. Intermediate datasets are data that has been created and used for intermediate steps when a model tool has been run.

Intermediate datasets can be accessed as long as the data format is compatible with the Python client library.

The following formats are supported (constants for these formats are in the `azureml.DataTypeIds` class):

- PlainText
- GenericCSV
- GenericTSV
- GenericCSVNoHeader
- GenericTSVNoHeader

You can determine the format by hovering over a module output node. It is displayed along with the node name, in a tooltip.

Some of the modules, such as the Split module, output to a format named `Dataset`, which is not supported by the Python client library.

You need to use a conversion module, such as Convert to CSV, to get an output into a supported format.

The following steps show an example that creates an experiment, runs it and accesses the intermediate dataset.

1. Create a new experiment.

2. Insert an **Adult Census Income Binary Classification dataset** module.

3. Insert a Split module, and connect its input to the dataset module output.

4. Insert a Convert to CSV module and connect its input to one of the Split module outputs.

5. Save the experiment, run it, and wait for the job to finish.

6. Click the output node on the Convert to CSV module.

7. When the context menu appears, select **Generate Data Access Code**.

8. Select the code snippet and copy it to your clipboard from the window that appears.

9. Paste the code in your notebook.

IP[y]: Notebook My Test Notebook Last Checkpoint: Jan 20 12:58 (unsaved changes)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

In [6]:
```
from azureml import Workspace
ws = Workspace(
    workspace_id='[redacted]',
    authorization_token='[redacted]'
)
experiment = ws.experiments['[redacted]']
ds = experiment.get_intermediate_dataset(
    node_id='13730c2f-489a-4c44-9f68-5eb8db2455e2-158',
    port_name='Results dataset',
    data_type_id='GenericCSV'
)
frame = ds.to_dataframe()
```

In [7]:
```
frame
```

Out[7]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss
0	52	Private	225317	5th-6th	3	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0
1	29	Private	154017	HS-grad	9	Never-married	Sales	Not-in-family	White	Female	0	0
2	25	Private	203570	HS-grad	9	Separated	Other-service	Unmarried	Black	Male	0	0

10. You can visualize the data using matplotlib. This displays in a histogram for the age column:

In [7]:
```
%matplotlib inline
```

In [8]:
```
frame.age.hist(grid=False, bins=6)
```

Out[8]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x9e427d0>
```

Use the Machine Learning Python client library to access, read, create, and manage datasets

Workspace

The workspace is the entry point for the Python client library. Provide the `Workspace` class with your workspace ID and authorization token to create an instance:

```
ws = Workspace(workspace_id='4c29e1adeba2e5a7cbeb0e4f4adfb4df',
               authorization_token='f4f3ade2c6aefdb1afb043cd8bcf3daf')
```

Enumerate datasets

To enumerate all datasets in a given workspace:

```
for ds in ws.datasets:
    print(ds.name)
```

To enumerate just the user-created datasets:

```
for ds in ws.user_datasets:
    print(ds.name)
```

To enumerate just the example datasets:

```
for ds in ws.example_datasets:
    print(ds.name)
```

You can access a dataset by name (which is case-sensitive):

```
ds = ws.datasets['my dataset name']
```

Or you can access it by index:

```
ds = ws.datasets[0]
```

Metadata

Datasets have metadata, in addition to content. (Intermediate datasets are an exception to this rule and do not have any metadata.)

Some metadata values are assigned by the user at creation time:

- `print(ds.name)`
- `print(ds.description)`
- `print(ds.family_id)`
- `print(ds.data_type_id)`

Others are values assigned by Azure ML:

- `print(ds.id)`
- `print(ds.created_date)`
- `print(ds.size)`

See the `SourceDataset` class for more on the available metadata.

Read contents

The code snippets provided by Machine Learning Studio (classic) automatically download and deserialize the dataset to a pandas DataFrame object. This is done with the `to_dataframe` method:

```
frame = ds.to_dataframe()
```

If you prefer to download the raw data, and perform the deserialization yourself, that is an option. At the moment, this is the only option for formats such as 'ARFF', which the Python client library cannot deserialize.

To read the contents as text:

```
text_data = ds.read_as_text()
```

To read the contents as binary:

```
binary_data = ds.read_as_binary()
```

You can also just open a stream to the contents:

```
with ds.open() as file:
    binary_data_chunk = file.read(1000)
```

Create a new dataset

The Python client library allows you to upload datasets from your Python program. These datasets are then available for use in your workspace.

If you have your data in a pandas DataFrame, use the following code:

```
from azureml import DataTypeIds

dataset = ws.datasets.add_from_dataframe(
    dataframe=frame,
    data_type_id=DataTypeIds.GenericCSV,
    name='my new dataset',
    description='my description'
)
```

If your data is already serialized, you can use:

```
from azureml import DataTypeIds

dataset = ws.datasets.add_from_raw_data(
    raw_data=raw_data,
    data_type_id=DataTypeIds.GenericCSV,
    name='my new dataset',
    description='my description'
)
```

The Python client library is able to serialize a pandas DataFrame to the following formats (constants for these are in the `azureml.DataTypeIds` class):

- PlainText
- GenericCSV
- GenericTSV
- GenericCSVNoHeader
- GenericTSVNoHeader

Update an existing dataset

If you try to upload a new dataset with a name that matches an existing dataset, you should get a conflict error.

To update an existing dataset, you first need to get a reference to the existing dataset:

```
dataset = ws.datasets['existing dataset']

print(dataset.data_type_id) # 'GenericCSV'
print(dataset.name)         # 'existing dataset'
print(dataset.description)  # 'data up to jan 2015'
```

Then use `update_from_dataframe` to serialize and replace the contents of the dataset on Azure:

```
dataset = ws.datasets['existing dataset']

dataset.update_from_dataframe(frame2)

print(dataset.data_type_id) # 'GenericCSV'
print(dataset.name)         # 'existing dataset'
print(dataset.description)  # 'data up to jan 2015'
```

If you want to serialize the data to a different format, specify a value for the optional `data_type_id` parameter.

```
from azureml import DataTypeIds

dataset = ws.datasets['existing dataset']

dataset.update_from_dataframe(
    dataframe=frame2,
    data_type_id=DataTypeIds.GenericTSV,
)

print(dataset.data_type_id) # 'GenericTSV'
print(dataset.name)         # 'existing dataset'
print(dataset.description)  # 'data up to jan 2015'
```

You can optionally set a new description by specifying a value for the `description` parameter.

```
dataset = ws.datasets['existing dataset']

dataset.update_from_dataframe(
    dataframe=frame2,
    description='data up to feb 2015',
)

print(dataset.data_type_id) # 'GenericCSV'
print(dataset.name)         # 'existing dataset'
print(dataset.description)  # 'data up to feb 2015'
```

You can optionally set a new name by specifying a value for the `name` parameter. From now on, you'll retrieve the dataset using the new name only. The following code updates the data, name, and description.

```
dataset = ws.datasets['existing dataset']

dataset.update_from_dataframe(
    dataframe=frame2,
    name='existing dataset v2',
    description='data up to feb 2015',
)

print(dataset.data_type_id)                    # 'GenericCSV'
print(dataset.name)                            # 'existing dataset v2'
print(dataset.description)                     # 'data up to feb 2015'

print(ws.datasets['existing dataset v2'].name) # 'existing dataset v2'
print(ws.datasets['existing dataset'].name)    # IndexError
```

The `data_type_id`, `name` and `description` parameters are optional and default to their previous value. The `dataframe` parameter is always required.

If your data is already serialized, use `update_from_raw_data` instead of `update_from_dataframe`. If you just pass in `raw_data` instead of `dataframe`, it works in a similar way.

Process Azure blob data with advanced analytics

11/19/2021 • 3 minutes to read • Edit Online

This document covers exploring data and generating features from data stored in Azure Blob storage.

Load the data into a Pandas data frame

In order to explore and manipulate a dataset, it must be downloaded from the blob source to a local file that can then be loaded in a Pandas data frame. Here are the steps to follow for this procedure:

1. Download the data from Azure blob with the following sample Python code using Blob service. Replace the variable in the code below with your specific values:

```
from azure.storage.blob import BlobService
import tables

STORAGEACCOUNTNAME= <storage_account_name>
STORAGEACCOUNTKEY= <storage_account_key>
LOCALFILENAME= <local_file_name>
CONTAINERNAME= <container_name>
BLOBNAME= <blob_name>

#download from blob
t1=time.time()
blob_service=BlobService(account_name=STORAGEACCOUNTNAME,account_key=STORAGEACCOUNTKEY)
blob_service.get_blob_to_path(CONTAINERNAME,BLOBNAME,LOCALFILENAME)
t2=time.time()
print(("It takes %s seconds to download "+blobname) % (t2 - t1))
```

2. Read the data into a Pandas data-frame from the downloaded file.

```
#LOCALFILE is the file path
dataframe_blobdata = pd.read_csv(LOCALFILE)
```

Now you are ready to explore the data and generate features on this dataset.

Data Exploration

Here are a few examples of ways to explore data using Pandas:

1. Inspect the number of rows and columns:

```
print 'the size of the data is: %d rows and  %d columns' % dataframe_blobdata.shape
```

2. Inspect the first or last few rows in the dataset as below:

```
dataframe_blobdata.head(10)

dataframe_blobdata.tail(10)
```

3. Check the data type each column was imported as using the following sample code

```
for col in dataframe_blobdata.columns:
    print dataframe_blobdata[col].name, ':\t', dataframe_blobdata[col].dtype
```

4. Check the basic stats for the columns in the data set as follows

```
dataframe_blobdata.describe()
```

5. Look at the number of entries for each column value as follows

```
dataframe_blobdata['<column_name>'].value_counts()
```

6. Count missing values versus the actual number of entries in each column using the following sample code

```
miss_num = dataframe_blobdata.shape[0] - dataframe_blobdata.count()
print miss_num
```

7. If you have missing values for a specific column in the data, you can drop them as follows:

```
dataframe_blobdata_noNA = dataframe_blobdata.dropna()
dataframe_blobdata_noNA.shape
```

   Another way to replace missing values is with the mode function:

```
dataframe_blobdata_mode =
dataframe_blobdata.fillna({'<column_name>':dataframe_blobdata['<column_name>'].mode()[0]})
```

8. Create a histogram plot using variable number of bins to plot the distribution of a variable:

```
dataframe_blobdata['<column_name>'].value_counts().plot(kind='bar')

np.log(dataframe_blobdata['<column_name>']+1).hist(bins=50)
```

9. Look at correlations between variables using a scatterplot or using the built-in correlation function:

```
#relationship between column_a and column_b using scatter plot
plt.scatter(dataframe_blobdata['<column_a>'], dataframe_blobdata['<column_b>'])

#correlation between column_a and column_b
dataframe_blobdata[['<column_a>', '<column_b>']].corr()
```

Feature Generation

We can generate features using Python as follows:

Indicator value-based Feature Generation

Categorical features can be created as follows:

1. Inspect the distribution of the categorical column:

```
dataframe_blobdata['<categorical_column>'].value_counts()
```

2. Generate indicator values for each of the column values:

```
#generate the indicator column
dataframe_blobdata_identity = pd.get_dummies(dataframe_blobdata['<categorical_column>'],
prefix='<categorical_column>_identity')
```

3. Join the indicator column with the original data frame:

```
#Join the dummy variables back to the original data frame
dataframe_blobdata_with_identity = dataframe_blobdata.join(dataframe_blobdata_identity)
```

4. Remove the original variable itself:

```
#Remove the original column rate_code in df1_with_dummy
dataframe_blobdata_with_identity.drop('<categorical_column>', axis=1, inplace=True)
```

Binning Feature Generation

For generating binned features, we proceed as follows:

1. Add a sequence of columns to bin a numeric column:

```
bins = [0, 1, 2, 4, 10, 40]
dataframe_blobdata_bin_id = pd.cut(dataframe_blobdata['<numeric_column>'], bins)
```

2. Convert binning to a sequence of boolean variables

```
dataframe_blobdata_bin_bool = pd.get_dummies(dataframe_blobdata_bin_id, prefix='<numeric_column>')
```

3. Finally, Join the dummy variables back to the original data frame

```
dataframe_blobdata_with_bin_bool = dataframe_blobdata.join(dataframe_blobdata_bin_bool)
```

Writing data back to Azure blob and consuming in Azure Machine Learning

After you have explored the data and created the necessary features, you can upload the data (sampled or featurized) to an Azure blob and consume it in Azure Machine Learning using the following steps: Additional features can be created in the Azure Machine Learning Studio (classic) as well.

1. Write the data frame to local file

```
dataframe.to_csv(os.path.join(os.getcwd(),LOCALFILENAME), sep='\t', encoding='utf-8', index=False)
```

2. Upload the data to Azure blob as follows:

```
from azure.storage.blob import BlobService
import tables

STORAGEACCOUNTNAME= <storage_account_name>
LOCALFILENAME= <local_file_name>
STORAGEACCOUNTKEY= <storage_account_key>
CONTAINERNAME= <container_name>
BLOBNAME= <blob_name>

output_blob_service=BlobService(account_name=STORAGEACCOUNTNAME,account_key=STORAGEACCOUNTKEY)
localfileprocessed = os.path.join(os.getcwd(),LOCALFILENAME) #assuming file is in current working
directory

try:

#perform upload
output_blob_service.put_block_blob_from_path(CONTAINERNAME,BLOBNAME,localfileprocessed)

except:
    print ("Something went wrong with uploading blob:"+BLOBNAME)
```

3. Now the data can be read from the blob using the Azure Machine Learning Import Data module as shown in the screen below:

Scalable data science with Azure Data Lake

11/19/2021 • 19 minutes to read • Edit Online

This walkthrough shows how to use Azure Data Lake to do data exploration and binary classification tasks on a sample of the NYC taxi trip and fare dataset to predict whether or not a tip is paid by a fare. It walks you through the steps of the Team Data Science Process, end-to-end, from data acquisition to model training, and then to the deployment of a web service that publishes the model.

Technologies

These technologies are used in this walkthrough.

- Azure Data Lake Analytics
- U-SQL and Visual Studio
- Python
- Azure Machine Learning
- Scripts

Azure Data Lake Analytics

The Microsoft Azure Data Lake has all the capabilities required to make it easy for data scientists to store data of any size, shape and speed, and to conduct data processing, advanced analytics, and machine learning modeling with high scalability in a cost-effective way. You pay on a per-job basis, only when data is actually being processed. Azure Data Lake Analytics includes U-SQL, a language that blends the declarative nature of SQL with the expressive power of C# to provide scalable distributed query capability. It enables you to process unstructured data by applying schema on read, insert custom logic and user-defined functions (UDFs), and includes extensibility to enable fine grained control over how to execute at scale. To learn more about the design philosophy behind U-SQL, see Visual Studio blog post.

Data Lake Analytics is also a key part of Cortana Analytics Suite and works with Azure Synapse Analytics, Power BI, and Data Factory. This combination gives you a complete cloud big data and advanced analytics platform.

This walkthrough begins by describing how to install the prerequisites and resources that are needed to complete data science process tasks. Then it outlines the data processing steps using U-SQL and concludes by showing how to use Python and Hive with Azure Machine Learning Studio (classic) to build and deploy the predictive models.

U-SQL and Visual Studio

This walkthrough recommends using Visual Studio to edit U-SQL scripts to process the dataset. The U-SQL scripts are described here and provided in a separate file. The process includes ingesting, exploring, and sampling the data. It also shows how to run a U-SQL scripted job from the Azure portal. Hive tables are created for the data in an associated HDInsight cluster to facilitate the building and deployment of a binary classification model in Azure Machine Learning Studio.

Python

This walkthrough also contains a section that shows how to build and deploy a predictive model using Python with Azure Machine Learning Studio. It provides a Jupyter notebook with the Python scripts for the steps in this process. The notebook includes code for some additional feature engineering steps and models construction such as multiclass classification and regression modeling in addition to the binary classification model outlined here. The regression task is to predict the amount of the tip based on other tip features.

Azure Machine Learning

Azure Machine Learning Studio (classic) is used to build and deploy the predictive models using two approaches: first with Python scripts and then with Hive tables on an HDInsight (Hadoop) cluster.

Scripts

Only the principal steps are outlined in this walkthrough. You can download the full **U-SQL script** and **Jupyter Notebook** from GitHub.

Prerequisites

Before you begin these topics, you must have the following:

- An Azure subscription. If you do not already have one, see Get Azure free trial.
- [Recommended] Visual Studio 2013 or later. If you do not already have one of these versions installed, you can download a free Community version from Visual Studio Community.

> **NOTE**
> Instead of Visual Studio, you can also use the Azure portal to submit Azure Data Lake queries. Instructions are provided on how to do so both with Visual Studio and on the portal in the section titled **Process data with U-SQL**.

Prepare data science environment for Azure Data Lake

To prepare the data science environment for this walkthrough, create the following resources:

- Azure Data Lake Storage (ADLS)
- Azure Data Lake Analytics (ADLA)
- Azure Blob storage account
- Azure Machine Learning Studio (classic) account
- Azure Data Lake Tools for Visual Studio (Recommended)

This section provides instructions on how to create each of these resources. If you choose to use Hive tables with Azure Machine Learning, instead of Python, to build a model, you also need to provision an HDInsight (Hadoop)

cluster. This alternative procedure in described in the Option 2 section.

> **NOTE**
>
> The **Azure Data Lake Store** can be created either separately or when you create the **Azure Data Lake Analytics** as the default storage. Instructions are referenced for creating each of these resources separately, but the Data Lake storage account need not be created separately.

Create an Azure Data Lake Storage

Create an ADLS from the Azure portal. For details, see Create an HDInsight cluster with Data Lake Store using Azure portal. Be sure to set up the Cluster AAD Identity in the **DataSource** blade of the **Optional Configuration** blade described there.

Create an Azure Data Lake Analytics account

Create an ADLA account from the Azure portal. For details, see Tutorial: get started with Azure Data Lake Analytics using Azure portal.

Create an Azure Blob storage account

Create an Azure Blob storage account from the Azure portal. For details, see the Create a storage account section in About Azure Storage accounts.

Set up an Azure Machine Learning Studio (classic) account

Sign up/into Azure Machine Learning Studio (classic) from the Azure Machine Learning studio page. Click on the **Get started now** button and then choose a "Free Workspace" or "Standard Workspace". Now your are ready to create experiments in Azure Machine Learning studio.

Install Azure Data Lake Tools [Recommended]

Install Azure Data Lake Tools for your version of Visual Studio from Azure Data Lake Tools for Visual Studio.

After the installation finishes, open up Visual Studio. You should see the Data Lake tab the menu at the top. Your Azure resources should appear in the left panel when you sign into your Azure account.

The NYC Taxi Trips dataset

The data set used here is a publicly available dataset -- the NYC Taxi Trips dataset. The NYC Taxi Trip data consists of about 20 GB of compressed CSV files (~48 GB uncompressed), recording more than 173 million individual trips and the fares paid for each trip. Each trip record includes the pickup and dropoff locations and times, anonymized hack (driver's) license number, and the medallion (taxi's unique ID) number. The data covers all trips in the year 2013 and is provided in the following two datasets for each month:

The 'trip_data' CSV contains trip details, such as number of passengers, pickup and dropoff points, trip duration, and trip length. Here are a few sample records:

```
medallion,hack_license,vendor_id,rate_code,store_and_fwd_flag,pickup_datetime,dropoff_datetime,passenger_count,trip_time_in_secs,trip_distance,pickup_longitude
89D227B655E5C82AECF13C3F540D4CF4,BA96DE419E711691B9445D6A6307C170,CMT,1,N,2013-01-01 15:11:48,2013-01-01 15:18:10,4,382,1.00,-73.978165,40.757977,-73.989838,40.751171
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,N,2013-01-06 00:18:35,2013-01-06 00:22:54,1,259,1.50,-74.006683,40.731781,-73.994499,40.75066
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,N,2013-01-05 18:49:41,2013-01-05 18:54:23,1,282,1.10,-74.004707,40.73777,-74.009834,40.726002
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,1,N,2013-01-07 23:54:15,2013-01-07 23:58:20,2,244,.70,-73.974602,40.759945,-73.984734,40.759388
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,1,N,2013-01-07 23:25:03,2013-01-07 23:34:24,1,560,2.10,-73.97625,40.748528,-74.002586,40.747868
```

The 'trip_fare' CSV contains details of the fare paid for each trip, such as payment type, fare amount, surcharge and taxes, tips and tolls, and the total amount paid. Here are a few sample records:

```
medallion, hack_license, vendor_id, pickup_datetime, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount
89D227B655E5C82AECF13C3F540D4CF4,BA96DE419E711691B9445D6A6307C170,CMT,2013-01-01 15:11:48,CSH,6.5,0,0.5,0,0,7
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,2013-01-06 00:18:35,CSH,6,0.5,0.5,0,0,7
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,2013-01-05 18:49:41,CSH,5.5,1,0.5,0,0,7
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,2013-01-07 23:54:15,CSH,5,0.5,0.5,0,0,6
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,2013-01-07 23:25:03,CSH,9.5,0.5,0.5,0,0,10.5
```

The unique key to join trip_data and trip_fare is composed of the following three fields: medallion, hack_license and pickup_datetime. The raw CSV files can be accessed from an Azure Storage blob. The U-SQL script for this join is in the Join trip and fare tables section.

Process data with U-SQL

The data processing tasks illustrated in this section include ingesting, checking quality, exploring, and sampling the data. How to join trip and fare tables is also shown. The final section shows run a U-SQL scripted job from the Azure portal. Here are links to each subsection:

- Data ingestion: read in data from public blob
- Data quality checks
- Data exploration
- Join trip and fare tables
- Data sampling
- Run U-SQL jobs

The U-SQL scripts are described here and provided in a separate file. You can download the full **U-SQL scripts** from GitHub.

To execute U-SQL, Open Visual Studio, click **File --> New --> Project**, choose **U-SQL Project**, name and save it to a folder.

> **NOTE**
> It is possible to use the Azure Portal to execute U-SQL instead of Visual Studio. You can navigate to the Azure Data Lake Analytics resource on the portal and submit queries directly as illustrated in the following figure:

Data Ingestion: Read in data from public blob

The location of the data in the Azure blob is referenced as **wasb://container_name@blob_storage_account_name.blob.core.windows.net/blob_name** and can be extracted using **Extractors.Csv()**. Substitute your own container name and storage account name in following scripts for container_name@blob_storage_account_name in the wasb address. Since the file names are in same format, it is possible to use **trip_data_{*}.csv** to read in all 12 trip files.

```
///Read in Trip data
@trip0 =
    EXTRACT
    medallion string,
    hack_license string,
    vendor_id string,
    rate_code string,
    store_and_fwd_flag string,
    pickup_datetime string,
    dropoff_datetime string,
    passenger_count string,
    trip_time_in_secs string,
    trip_distance string,
    pickup_longitude string,
    pickup_latitude string,
    dropoff_longitude string,
    dropoff_latitude string
// This is reading 12 trip data from blob
FROM "wasb://container_name@blob_storage_account_name.blob.core.windows.net/nyctaxitrip/trip_data_{*}.csv"
USING Extractors.Csv();
```

Since there are headers in the first row, you need to remove the headers and change column types into appropriate ones. You can either save the processed data to Azure Data Lake Storage using **swebhdfs://data_lake_storage_name.azuredatalakestorage.net/folder_name/file_name_** or to Azure Blob storage account using **wasb://container_name@blob_storage_account_name.blob.core.windows.net/blob_name**.

```
// change data types
@trip =
    SELECT
    medallion,
    hack_license,
    vendor_id,
    rate_code,
    store_and_fwd_flag,
    DateTime.Parse(pickup_datetime) AS pickup_datetime,
    DateTime.Parse(dropoff_datetime) AS dropoff_datetime,
    Int32.Parse(passenger_count) AS passenger_count,
    Double.Parse(trip_time_in_secs) AS trip_time_in_secs,
    Double.Parse(trip_distance) AS trip_distance,
    (pickup_longitude==string.Empty ? 0: float.Parse(pickup_longitude)) AS pickup_longitude,
    (pickup_latitude==string.Empty ? 0: float.Parse(pickup_latitude)) AS pickup_latitude,
    (dropoff_longitude==string.Empty ? 0: float.Parse(dropoff_longitude)) AS dropoff_longitude,
    (dropoff_latitude==string.Empty ? 0: float.Parse(dropoff_latitude)) AS dropoff_latitude
FROM @trip0
WHERE medallion != "medallion";

////output data to ADL
OUTPUT @trip
TO "swebhdfs://data_lake_storage_name.azuredatalakestore.net/nyctaxi_folder/demo_trip.csv"
USING Outputters.Csv();

////Output data to blob
OUTPUT @trip
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_trip.csv"
USING Outputters.Csv();
```

Similarly you can read in the fare data sets. Right-click Azure Data Lake Storage, you can choose to look at your data in **Azure portal --> Data Explorer** or **File Explorer** within Visual Studio.

Data quality checks

After trip and fare tables have been read in, data quality checks can be done in the following way. The resulting CSV files can be output to Azure Blob storage or Azure Data Lake Storage.

Find the number of medallions and unique number of medallions:

```
///check the number of medallions and unique number of medallions
@trip2 =
    SELECT
    medallion,
    vendor_id,
    pickup_datetime.Month AS pickup_month
    FROM @trip;

@ex_1 =
    SELECT
    pickup_month,
    COUNT(medallion) AS cnt_medallion,
    COUNT(DISTINCT(medallion)) AS unique_medallion
    FROM @trip2
    GROUP BY pickup_month;
    OUTPUT @ex_1
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_1.csv"
USING Outputters.Csv();
```

Find those medallions that had more than 100 trips:

```
///find those medallions that had more than 100 trips
@ex_2 =
    SELECT medallion,
           COUNT(medallion) AS cnt_medallion
    FROM @trip2
    //where pickup_datetime >= "2013-01-01t00:00:00.0000000" and pickup_datetime <= "2013-04-
01t00:00:00.0000000"
    GROUP BY medallion
    HAVING COUNT(medallion) > 100;
    OUTPUT @ex_2
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_2.csv"
USING Outputters.Csv();
```

Find those invalid records in terms of pickup_longitude:

```
///find those invalid records in terms of pickup_longitude
@ex_3 =
    SELECT COUNT(medallion) AS cnt_invalid_pickup_longitude
    FROM @trip
    WHERE
    pickup_longitude <- 90 OR pickup_longitude > 90;
    OUTPUT @ex_3
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_3.csv"
USING Outputters.Csv();
```

Find missing values for some variables:

```
//check missing values
@res =
    SELECT *,
           (medallion == null? 1 : 0) AS missing_medallion
    FROM @trip;

@trip_summary6 =
    SELECT
        vendor_id,
    SUM(missing_medallion) AS medallion_empty,
    COUNT(medallion) AS medallion_total,
    COUNT(DISTINCT(medallion)) AS medallion_total_unique
    FROM @res
    GROUP BY vendor_id;
OUTPUT @trip_summary6
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_16.csv"
USING Outputters.Csv();
```

Data exploration

Do some data exploration with the following scripts to get a better understanding of the data.

Find the distribution of tipped and non-tipped trips:

```
///tipped vs. not tipped distribution
@tip_or_not =
    SELECT *,
           (tip_amount > 0 ? 1: 0) AS tipped
    FROM @fare;

@ex_4 =
    SELECT tipped,
           COUNT(*) AS tip_freq
    FROM @tip_or_not
    GROUP BY tipped;
    OUTPUT @ex_4
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_4.csv"
USING Outputters.Csv();
```

Find the distribution of tip amount with cut-off values: 0, 5, 10, and 20 dollars.

```
//tip class/range distribution
@tip_class =
    SELECT *,
           (tip_amount >20? 4: (tip_amount >10? 3:(tip_amount >5 ? 2:(tip_amount > 0 ? 1: 0)))) AS tip_class
    FROM @fare;
@ex_5 =
    SELECT tip_class,
           COUNT(*) AS tip_freq
    FROM @tip_class
    GROUP BY tip_class;
    OUTPUT @ex_5
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_5.csv"
USING Outputters.Csv();
```

Find basic statistics of trip distance:

```
// find basic statistics for trip_distance
@trip_summary4 =
    SELECT
        vendor_id,
        COUNT(*) AS cnt_row,
        MIN(trip_distance) AS min_trip_distance,
        MAX(trip_distance) AS max_trip_distance,
        AVG(trip_distance) AS avg_trip_distance
    FROM @trip
    GROUP BY vendor_id;
OUTPUT @trip_summary4
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_14.csv"
USING Outputters.Csv();
```

Find the percentiles of trip distance:

```
// find percentiles of trip_distance
@trip_summary3 =
    SELECT DISTINCT vendor_id AS vendor,
                    PERCENTILE_DISC(0.25) WITHIN GROUP(ORDER BY trip_distance) OVER(PARTITION BY vendor_id)
AS median_trip_distance_disc,
                    PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY trip_distance) OVER(PARTITION BY vendor_id)
AS median_trip_distance_disc,
                    PERCENTILE_DISC(0.75) WITHIN GROUP(ORDER BY trip_distance) OVER(PARTITION BY vendor_id)
AS median_trip_distance_disc
    FROM @trip;
   // group by vendor_id;
OUTPUT @trip_summary3
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_13.csv"
USING Outputters.Csv();
```

Join trip and fare tables

Trip and fare tables can be joined by medallion, hack_license, and pickup_time.

```
//join trip and fare table

@model_data_full =
SELECT t.*,
f.payment_type, f.fare_amount, f.surcharge, f.mta_tax, f.tolls_amount,  f.total_amount, f.tip_amount,
(f.tip_amount > 0 ? 1: 0) AS tipped,
(f.tip_amount >20? 4: (f.tip_amount >10? 3:(f.tip_amount >5 ? 2:(f.tip_amount > 0 ? 1: 0)))) AS tip_class
FROM @trip AS t JOIN  @fare AS f
ON   (t.medallion == f.medallion AND t.hack_license == f.hack_license AND t.pickup_datetime ==
f.pickup_datetime)
WHERE   (pickup_longitude != 0 AND dropoff_longitude != 0 );

//// output to blob
OUTPUT @model_data_full
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_7_full_data.csv"
USING Outputters.Csv();

////output data to ADL
OUTPUT @model_data_full
TO "swebhdfs://data_lake_storage_name.azuredatalakestore.net/nyctaxi_folder/demo_ex_7_full_data.csv"
USING Outputters.Csv();
```

For each level of passenger count, calculate the number of records, average tip amount, variance of tip amount, percentage of tipped trips.

```
// contingency table
@trip_summary8 =
    SELECT passenger_count,
           COUNT(*) AS cnt,
           AVG(tip_amount) AS avg_tip_amount,
           VAR(tip_amount) AS var_tip_amount,
           SUM(tipped) AS cnt_tipped,
           (float)SUM(tipped)/COUNT(*) AS pct_tipped
    FROM @model_data_full
    GROUP BY passenger_count;
    OUTPUT @trip_summary8
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_17.csv"
USING Outputters.Csv();
```

Data sampling

First, randomly select 0.1% of the data from the joined table:

```
//random select 1/1000 data for modeling purpose
@addrownumberres_randomsample =
SELECT *,
        ROW_NUMBER() OVER() AS rownum
FROM @model_data_full;

@model_data_random_sample_1_1000 =
SELECT *
FROM @addrownumberres_randomsample
WHERE rownum % 1000 == 0;

OUTPUT @model_data_random_sample_1_1000
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_7_random_1_1000.csv"
USING Outputters.Csv();
```

Then do stratified sampling by binary variable tip_class:

```
//stratified random select 1/1000 data for modeling purpose
@addrownumberres_stratifiedsample =
SELECT *,
        ROW_NUMBER() OVER(PARTITION BY tip_class) AS rownum
FROM @model_data_full;

@model_data_stratified_sample_1_1000 =
SELECT *
FROM @addrownumberres_stratifiedsample
WHERE rownum % 1000 == 0;
//// output to blob
OUTPUT @model_data_stratified_sample_1_1000
TO "wasb://container_name@blob_storage_account_name.blob.core.windows.net/demo_ex_9_stratified_1_1000.csv"
USING Outputters.Csv();
////output data to ADL
OUTPUT @model_data_stratified_sample_1_1000
TO "swebhdfs://data_lake_storage_name.azuredatalakestore.net/nyctaxi_folder/demo_ex_9_stratified_1_1000.csv"
USING Outputters.Csv();
```

Run U-SQL jobs

After editing U-SQL scripts, you can submit them to the server using your Azure Data Lake Analytics account. Click **Data Lake**, **Submit Job**, select your **Analytics Account**, choose **Parallelism**, and click **Submit** button.

When the job is complied successfully, the status of your job is displayed in Visual Studio for monitoring. After the job completes, you can even replay the job execution process and find out the bottleneck steps to improve your job efficiency. You can also go to Azure portal to check the status of your U-SQL jobs.

Now you can check the output files in either Azure Blob storage or Azure portal. Use the stratified sample data for our modeling in the next step.

Build and deploy models in Azure Machine Learning

Two options are available for you to pull data into Azure Machine Learning to build and

- In the first option, you use the sampled data that has been written to an Azure Blob (in the **Data sampling** step above) and use Python to build and deploy models from Azure Machine Learning.
- In the second option, you query the data in Azure Data Lake directly using a Hive query. This option requires that you create a new HDInsight cluster or use an existing HDInsight cluster where the Hive tables point to the NY Taxi data in Azure Data Lake Storage. Both these options are discussed in the following sections.

Option 1: Use Python to build and deploy machine learning models

To build and deploy machine learning models using Python, create a Jupyter Notebook on your local machine or in Azure Machine Learning Studio. The Jupyter Notebook provided on GitHub contains the full code to explore, visualize data, feature engineering, modeling, and deployment. In this article, just the modeling and deployment are covered.

Import Python libraries

In order to run the sample Jupyter Notebook or the Python script file, the following Python packages are needed. If you are using the Azure Machine Learning Notebook service, these packages have been pre-installed.

```
import pandas as pd
from pandas import Series, DataFrame
import numpy as np
import matplotlib.pyplot as plt
from time import time
import pyodbc
import os
from azure.storage.blob import BlobService
import tables
import time
import zipfile
import random
import sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import train_test_split
from sklearn import metrics
from __future__ import division
from sklearn import linear_model
from azureml import services
```

Read in the data from blob

- Connection String

```
CONTAINERNAME = 'test1'
STORAGEACCOUNTNAME = 'XXXXXXXXX'
STORAGEACCOUNTKEY = 'YYYYYYYYYYYYYYYYYYYYYYYYYYYY'
BLOBNAME = 'demo_ex_9_stratified_1_1000_copy.csv'
blob_service = BlobService(account_name=STORAGEACCOUNTNAME,account_key=STORAGEACCOUNTKEY)
```

- Read in as text

```
t1 = time.time()
data = blob_service.get_blob_to_text(CONTAINERNAME,BLOBNAME).split("\n")
t2 = time.time()
print(("It takes %s seconds to read in "+BLOBNAME) % (t2 - t1))
```

```
It takes 1.61118912697 seconds to read in demo_ex_9_stratified_1_1000_copy.csv
```

- Add column names and separate columns

```
colnames =
['medallion','hack_license','vendor_id','rate_code','store_and_fwd_flag','pickup_datetime','dropoff_d
atetime',
'passenger_count','trip_time_in_secs','trip_distance','pickup_longitude','pickup_latitude','dropoff_l
ongitude','dropoff_latitude',
'payment_type', 'fare_amount', 'surcharge', 'mta_tax', 'tolls_amount',  'total_amount', 'tip_amount',
'tipped', 'tip_class', 'rownum']
df1 = pd.DataFrame([sub.split(",") for sub in data], columns = colnames)
```

- Change some columns to numeric

```
cols_2_float =
['trip_time_in_secs','pickup_longitude','pickup_latitude','dropoff_longitude','dropoff_latitude',
'fare_amount', 'surcharge','mta_tax','tolls_amount','total_amount','tip_amount',
'passenger_count','trip_distance'
,'tipped','tip_class','rownum']
for col in cols_2_float:
    df1[col] = df1[col].astype(float)
```

Build machine learning models

Here you build a binary classification model to predict whether a trip is tipped or not. In the Jupyter Notebook you can find other two models: multiclass classification, and regression models.

- First you need to create dummy variables that can be used in scikit-learn models

```
df1_payment_type_dummy = pd.get_dummies(df1['payment_type'], prefix='payment_type_dummy')
df1_vendor_id_dummy = pd.get_dummies(df1['vendor_id'], prefix='vendor_id_dummy')
```

- Create data frame for the modeling

```
cols_to_keep = ['tipped', 'trip_distance', 'passenger_count']
data = df1[cols_to_keep].join([df1_payment_type_dummy,df1_vendor_id_dummy])

X = data.iloc[:,1:]
Y = data.tipped
```

- Training and testing 60-40 split

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4, random_state=0)
```

- Logistic Regression in training set

```
model = LogisticRegression()
logit_fit = model.fit(X_train, Y_train)
print ('Coefficients: \n', logit_fit.coef_)
Y_train_pred = logit_fit.predict(X_train)
```

```
('Coefficients: \n', array([[-0.05133172, -0.008217  ,  5.45355932, -6.63170116, -1.79058312,
        -2.79956737,  4.60387707, -0.33341385, -0.83100141]]))
```

- Score testing data set

```
Y_test_pred = logit_fit.predict(X_test)
```

- Calculate Evaluation metrics

```
fpr_train, tpr_train, thresholds_train = metrics.roc_curve(Y_train, Y_train_pred)
print fpr_train, tpr_train, thresholds_train

fpr_test, tpr_test, thresholds_test = metrics.roc_curve(Y_test, Y_test_pred)
print fpr_test, tpr_test, thresholds_test

#AUC
print metrics.auc(fpr_train,tpr_train)
print metrics.auc(fpr_test,tpr_test)

#Confusion Matrix
print metrics.confusion_matrix(Y_train,Y_train_pred)
print metrics.confusion_matrix(Y_test,Y_test_pred)
```

```
[ 0.          0.03372081  1.        ] [ 0.          0.99994397  1.        ] [ 2.  1.  0.]
[ 0.          0.03342893  1.        ] [ 0.          0.99983179  1.        ] [ 2.  1.  0.]
0.983111577209
0.983201427013
[[46966  1639]
 [    3 53535]]
[[31343  1084]
 [    6 35663]]
```

Build Web Service API and consume it in Python

You want to operationalize the machine learning model after it has been built. The binary logistic model is used here as an example. Make sure the scikit-learn version in your local machine is 0.15.1 (Azure Machine Learning Studio is already at least at this version).

- Find your workspace credentials from Azure Machine Learning Studio (classic) settings. In Azure Machine Learning Studio, click **Settings** --> **Name** --> **Authorization Tokens**.

```
workspaceid = 'xxxxxxxxxxxxxxxxxxxxxxxxxx'
auth_token = 'xxxxxxxxxxxxxxxxxxxxxxxxxx'
```

- Create Web Service

```
@services.publish(workspaceid, auth_token)
@services.types(trip_distance = float, passenger_count = float, payment_type_dummy_CRD = float,
payment_type_dummy_CSH=float, payment_type_dummy_DIS = float, payment_type_dummy_NOC = float,
payment_type_dummy_UNK = float, vendor_id_dummy_CMT = float, vendor_id_dummy_VTS = float)
@services.returns(int) #0, or 1
def predictNYCTAXI(trip_distance, passenger_count, payment_type_dummy_CRD,
payment_type_dummy_CSH,payment_type_dummy_DIS, payment_type_dummy_NOC, payment_type_dummy_UNK,
vendor_id_dummy_CMT, vendor_id_dummy_VTS ):
    inputArray = [trip_distance, passenger_count, payment_type_dummy_CRD, payment_type_dummy_CSH,
payment_type_dummy_DIS, payment_type_dummy_NOC, payment_type_dummy_UNK, vendor_id_dummy_CMT,
vendor_id_dummy_VTS]
    return logit_fit.predict(inputArray)
```

- Get web service credentials

```
url = predictNYCTAXI.service.url
api_key =  predictNYCTAXI.service.api_key

print url
print api_key

@services.service(url, api_key)
@services.types(trip_distance = float, passenger_count = float, payment_type_dummy_CRD = float,
payment_type_dummy_CSH=float,payment_type_dummy_DIS = float, payment_type_dummy_NOC = float,
payment_type_dummy_UNK = float, vendor_id_dummy_CMT = float, vendor_id_dummy_VTS = float)
@services.returns(float)
def NYCTAXIPredictor(trip_distance, passenger_count, payment_type_dummy_CRD,
payment_type_dummy_CSH,payment_type_dummy_DIS, payment_type_dummy_NOC, payment_type_dummy_UNK,
vendor_id_dummy_CMT, vendor_id_dummy_VTS ):
    pass
```

- Call Web service API. Typically, wait 5-10 seconds after the previous step.

```
NYCTAXIPredictor(1,2,1,0,0,0,0,0,1)
```

```
1
```

Option 2: Create and deploy models directly in Azure Machine Learning

Azure Machine Learning Studio (classic) can read data directly from Azure Data Lake Storage and then be used to create and deploy models. This approach uses a Hive table that points at the Azure Data Lake Storage. A separate Azure HDInsight cluster needs to be provisioned for the Hive table.

Create an HDInsight Linux Cluster

Create an HDInsight Cluster (Linux) from the Azure portal. For details, see the **Create an HDInsight cluster with access to Azure Data Lake Storage** section in Create an HDInsight cluster with Data Lake Store using Azure portal.

Create Hive table in HDInsight

Now you create Hive tables to be used in Azure Machine Learning Studio (classic) in the HDInsight cluster using the data stored in Azure Data Lake Storage in the previous step. Go to the HDInsight cluster created. Click **Settings** --> **Properties** --> **Cluster AAD Identity** --> **ADLS Access**, make sure your Azure Data Lake Storage account is added in the list with read, write, and execute rights.

Then click **Dashboard** next to the **Settings** button and a window pops up. Click **Hive View** in the upper right corner of the page and you should see the **Query Editor**.

Paste in the following Hive scripts to create a table. The location of data source is in Azure Data Lake Storage reference in this way: **adl://data_lake_store_name.azuredatalakestore.net:443/folder_name/file_name**.

```
CREATE EXTERNAL TABLE nyc_stratified_sample
(
    medallion string,
    hack_license string,
    vendor_id string,
    rate_code string,
    store_and_fwd_flag string,
    pickup_datetime string,
    dropoff_datetime string,
    passenger_count string,
    trip_time_in_secs string,
    trip_distance string,
    pickup_longitude string,
    pickup_latitude string,
    dropoff_longitude string,
    dropoff_latitude string,
  payment_type string,
  fare_amount string,
  surcharge string,
  mta_tax string,
  tolls_amount string,
  total_amount string,
  tip_amount string,
  tipped string,
  tip_class string,
  rownum string
  )
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' lines terminated by '\n'
LOCATION
'adl://data_lake_storage_name.azuredatalakestore.net:443/nyctaxi_folder/demo_ex_9_stratified_1_1000_copy.csv
';
```

When the query completes, you should see the results like this:

Query Process Results (Status: Succeeded)

Save results...

Logs | Results

Filter columns...

previous next

nyc_stratified_sample_weig.medallion	nyc_stratified_sample_weig.hack_license	nyc_stratified_sample_we
"0053334C798EC6C8E637657962030F99"	"9EF690115D60940E7A8039A67542642E"	"VTS"
"00B99071EE4DC8266384113B91E6AC13"	"081B28EDA7F73E5E2C97573F0DBAC25D"	"CMT"
"011E4CBA1987A8553F8EA5681FFBF7F1"	"F53B26859F6C46C24E39EEAEE8096268"	"CMT"
"1109955CCAABCBCE1A22BCED5F1DBFF5"	"EAA69F43239E5C6609CD8FF4D6725836"	"CMT"
"0A415B814A6479EE706972D2FD6DA08A"	"12A32F655B8C9CE3AC7872477A641974"	"VTS"
"02B29197FB7470B583ED12167D19E998"	"C1EEBC8298A619F86637D7E97F6BDD5C"	"CMT"
"03055B956C21B1F915DCDB118AA79F21"	"E0C7A67293DE535DB04F8AFD8BF28F73"	"VTS"
"037673EEAE0DCB912D06BED04E89D89D"	"3B247BD1230E95A0D9FED3E47FECB8D9"	"CMT"
"03BF54085C92C385889B957D804780D1"	"776D5633A2041CEBDE03092E401D61DB"	"CMT"
"0437660BC3704C2F185301D539434A64"	"AE9AB8C79A2A0EB6DDA76B7024FF6029"	"CMT"

Build and deploy models in Azure Machine Learning Studio

You are now ready to build and deploy a model that predicts whether or not a tip is paid with Azure Machine Learning. The stratified sample data is ready to be used in this binary classification (tip or not) problem. The predictive models using multiclass classification (tip_class) and regression (tip_amount) can also be built and deployed with Azure Machine Learning Studio, but here it is only shown how to handle the case using the binary classification model.

1. Get the data into Azure Machine Learning Studio (classic) using the **Import Data** module, available in the **Data Input and Output** section. For more information, see the Import Data module reference page.
2. Select **Hive Query** as the **Data source** in the **Properties** panel.
3. Paste the following Hive script in the **Hive database query** editor

```
select * from nyc_stratified_sample;
```

4. Enter the URI of HDInsight cluster (this URI can be found in Azure portal), Hadoop credentials, location of output data, and Azure Storage account name/key/container name.

An example of a binary classification experiment reading data from Hive table is shown in the following figure:

After the experiment is created, click **Set Up Web Service** --> **Predictive Web Service**

Run the automatically created scoring experiment, when it finishes, click **Deploy Web Service**

The web service dashboard displays shortly:

Summary

By completing this walkthrough, you have created a data science environment for building scalable end-to-end solutions in Azure Data Lake. This environment was used to analyze a large public dataset, taking it through the canonical steps of the Data Science Process, from data acquisition through model training, and then to the deployment of the model as a web service. U-SQL was used to process, explore, and sample the data. Python and Hive were used with Azure Machine Learning Studio (classic) to build and deploy predictive models.

What's next?

The learning path for the Team Data Science Process (TDSP) provides links to topics describing each step in the advanced analytics process. There are a series of walkthroughs itemized on the Team Data Science Process walkthroughs page that showcase how to use resources and services in various predictive analytics scenarios:

- The Team Data Science Process in action: using Azure Synapse Analytics
- The Team Data Science Process in action: using HDInsight Hadoop clusters
- The Team Data Science Process: using SQL Server
- Overview of the Data Science Process using Spark on Azure HDInsight

Process data in a SQL Server virtual machine on Azure

11/19/2021 • 6 minutes to read • Edit Online

This document covers how to explore data and generate features for data stored in a SQL Server VM on Azure. This goal may be completed by data wrangling using SQL or by using a programming language like Python.

> **NOTE**
> The sample SQL statements in this document assume that data is in SQL Server. If it isn't, refer to the cloud data science process map to learn how to move your data to SQL Server.

Using SQL

We describe the following data wrangling tasks in this section using SQL:

1. Data Exploration
2. Feature Generation

Data Exploration

Here are a few sample SQL scripts that can be used to explore data stores in SQL Server.

> **NOTE**
> For a practical example, you can use the NYC Taxi dataset and refer to the IPNB titled NYC Data wrangling using IPython Notebook and SQL Server for an end-to-end walk-through.

1. Get the count of observations per day

```
SELECT CONVERT(date, <date_columnname>) as date, count(*) as c from <tablename> group by CONVERT(date,
<date_columnname>)
```

2. Get the levels in a categorical column

```
select distinct <column_name> from <databasename>
```

3. Get the number of levels in combination of two categorical columns

```
select <column_a>, <column_b>,count(*) from <tablename> group by <column_a>, <column_b>
```

4. Get the distribution for numerical columns

```
select <column_name>, count(*) from <tablename> group by <column_name>
```

Feature Generation

In this section, we describe ways of generating features using SQL:

1. Count based Feature Generation
2. Binning Feature Generation
3. Rolling out the features from a single column

> **NOTE**
> Once you generate additional features, you can either add them as columns to the existing table or create a new table with the additional features and primary key, that can be joined with the original table.

Count based Feature Generation

The following examples demonstrate two ways of generating count features. The first method uses conditional sum and the second method uses the 'where' clause. These results may then be joined with the original table (using primary key columns) to have count features alongside the original data.

```
select <column_name1>,<column_name2>,<column_name3>, COUNT(*) as Count_Features from <tablename> group by
<column_name1>,<column_name2>,<column_name3>

select <column_name1>,<column_name2> , sum(1) as Count_Features from <tablename>
where <column_name3> = '<some_value>' group by <column_name1>,<column_name2>
```

Binning Feature Generation

The following example shows how to generate binned features by binning (using five bins) a numerical column that can be used as a feature instead:

```
SELECT <column_name>, NTILE(5) OVER (ORDER BY <column_name>) AS BinNumber from <tablename>
```

Rolling out the features from a single column

In this section, we demonstrate how to roll out a single column in a table to generate additional features. The example assumes that there is a latitude or longitude column in the table from which you are trying to generate features.

Here is a brief primer on latitude/longitude location data (resourced from stackoverflow How to measure the accuracy of latitude and longitude?). This guidance is useful to understand before including location as one or more features:

- The sign tells us whether we are north or south, east or west on the globe.
- A nonzero hundreds digit tells us that we're using longitude, not latitude!
- The tens digit gives a position to about 1,000 kilometers. It gives us useful information about what continent or ocean we are on.
- The units digit (one decimal degree) gives a position up to 111 kilometers (60 nautical miles, about 69 miles). It can tell you roughly what state, country, or region you're in.
- The first decimal place is worth up to 11.1 km: it can distinguish the position of one large city from a neighboring large city.
- The second decimal place is worth up to 1.1 km: it can separate one village from the next.
- The third decimal place is worth up to 110 m: it can identify a large agricultural field or institutional campus.
- The fourth decimal place is worth up to 11 m: it can identify a parcel of land. It is comparable to the typical accuracy of an uncorrected GPS unit with no interference.
- The fifth decimal place is worth up to 1.1 m: it distinguishes trees from each other. Accuracy to this level with commercial GPS units can only be achieved with differential correction.
- The sixth decimal place is worth up to 0.11 m: you can use this for laying out structures in detail, for designing landscapes, building roads. It should be more than good enough for tracking movements of glaciers and rivers. This can be achieved by taking painstaking measures with GPS, such as differentially corrected GPS.

The location information can be featurized as follows, separating out region, location, and city information. You

can also call a REST end point such as Bing Maps API available at Find a Location by Point to get the region/district information.

```
select
    <location_columnname>
    ,round(<location_columnname>,0) as l1
    ,l2=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 1 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),1,1) else '0' end
    ,l3=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 2 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),2,1) else '0' end
    ,l4=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 3 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),3,1) else '0' end
    ,l5=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 4 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),4,1) else '0' end
    ,l6=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 5 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),5,1) else '0' end
    ,l7=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 6 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),6,1) else '0' end
from <tablename>
```

These location-based features can be further used to generate additional count features as described earlier.

> **TIP**
>
> You can programmatically insert the records using your language of choice. You may need to insert the data in chunks to improve write efficiency (for an example of how to do this using pyodbc, see A HelloWorld sample to access SQLServer with python). Another alternative is to insert data in the database using the BCP utility.

Connecting to Azure Machine Learning

The newly generated feature can be added as a column to an existing table or stored in a new table and joined with the original table for machine learning. Features can be generated or accessed if already created, using the Import Data module in Azure Machine Learning as shown below:

Using a programming language like Python

Using Python to explore data and generate features when the data is in SQL Server is similar to processing data in Azure blob using Python as documented in Process Azure Blob data in your data science environment. Load the data from the database into a pandas data frame for more processing. We document the process of connecting to the database and loading the data into the data frame in this section.

The following connection string format can be used to connect to a SQL Server database from Python using pyodbc (replace servername, dbname, username, and password with your specific values):

```
#Set up the SQL Azure connection
import pyodbc
conn = pyodbc.connect('DRIVER={SQL Server};SERVER=<servername>;DATABASE=<dbname>;UID=<username>;PWD=
<password>')
```

The Pandas library in Python provides a rich set of data structures and data analysis tools for data manipulation for Python programming. The code below reads the results returned from a SQL Server database into a Pandas data frame:

```
# Query database and load the returned results in pandas data frame
data_frame = pd.read_sql('''select <columnname1>, <columnname2>... from <tablename>''', conn)
```

Now you can work with the Pandas data frame as covered in the article Process Azure Blob data in your data science environment.

Azure Data Science in Action Example

For an end-to-end walkthrough example of the Azure Data Science Process using a public dataset, see Azure Data Science Process in Action.

Cheat sheet for an automated data pipeline for Azure Machine Learning predictions

11/19/2021 • 2 minutes to read • Edit Online

The **Microsoft Azure Machine Learning automated data pipeline cheat sheet** helps you navigate through the technology you can use to get your data to your Machine Learning web service where it can be scored by your predictive analytics model.

Depending on whether your data is on-premises, in the cloud, or real-time streaming, there are different mechanisms available to move the data to your web service endpoint for scoring. This cheat sheet walks you through the decisions you need to make, and it offers links to articles that can help you develop your solution.

Download the Machine Learning automated data pipeline cheat sheet

Once you download the cheat sheet, you can print it in tabloid size (11 x 17 in.).

Download the cheat sheet here: **Microsoft Azure Machine Learning automated data pipeline cheat sheet**

More help with Machine Learning Studio

- For an overview of Microsoft Azure Machine Learning, see Introduction to machine learning on Microsoft Azure.
- For an explanation of how to deploy a scoring web service, see Deploy an Azure Machine Learning web service.
- For a discussion of how to consume a scoring web service, see How to consume an Azure Machine Learning Web service.

Data science using Spark on Azure HDInsight

11/19/2021 • 9 minutes to read • Edit Online

This suite of topics shows how to use HDInsight Spark to complete common data science tasks such as data ingestion, feature engineering, modeling, and model evaluation. The data used is a sample of the 2013 NYC taxi trip and fare dataset. The models built include logistic and linear regression, random forests, and gradient boosted trees. The topics also show how to store these models in Azure blob storage (WASB) and how to score and evaluate their predictive performance. More advanced topics cover how models can be trained using cross-validation and hyper-parameter sweeping. This overview topic also references the topics that describe how to set up the Spark cluster that you need to complete the steps in the walkthroughs provided.

Spark and MLlib

Spark is an open-source parallel processing framework that supports in-memory processing to boost the performance of big-data analytic applications. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory distributed computation capabilities make it a good choice for the iterative algorithms used in machine learning and graph computations. MLlib is Spark's scalable machine learning library that brings the algorithmic modeling capabilities to this distributed environment.

HDInsight Spark

HDInsight Spark is the Azure hosted offering of open-source Spark. It also includes support for **Jupyter PySpark notebooks** on the Spark cluster that can run Spark SQL interactive queries for transforming, filtering, and visualizing data stored in Azure Blobs (WASB). PySpark is the Python API for Spark. The code snippets that provide the solutions and show the relevant plots to visualize the data here run in Jupyter notebooks installed on the Spark clusters. The modeling steps in these topics contain code that shows how to train, evaluate, save, and consume each type of model.

Setup: Spark clusters and Jupyter notebooks

Setup steps and code are provided in this walkthrough for using an HDInsight Spark 1.6. But Jupyter notebooks are provided for both HDInsight Spark 1.6 and Spark 2.0 clusters. A description of the notebooks and links to them are provided in the Readme.md for the GitHub repository containing them. Moreover, the code here and in the linked notebooks is generic and should work on any Spark cluster. If you are not using HDInsight Spark, the cluster setup and management steps may be slightly different from what is shown here. For convenience, here are the links to the Jupyter notebooks for Spark 1.6 (to be run in the pySpark kernel of the Jupyter Notebook server) and Spark 2.0 (to be run in the pySpark3 kernel of the Jupyter Notebook server):

Spark 1.6 notebooks

These notebooks are to be run in the pySpark kernel of Jupyter notebook server.

- pySpark-machine-learning-data-science-spark-data-exploration-modeling.ipynb: Provides information on how to perform data exploration, modeling, and scoring with several different algorithms.
- pySpark-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb: Includes topics in notebook #1, and model development using hyperparameter tuning and cross-validation.
- pySpark-machine-learning-data-science-spark-model-consumption.ipynb: Shows how to operationalize a saved model using Python on HDInsight clusters.

Spark 2.0 notebooks

These notebooks are to be run in the pySpark3 kernel of Jupyter notebook server.

- Spark2.0-pySpark3-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb: This file provides information on how to perform data exploration, modeling, and scoring in Spark 2.0 clusters using the NYC Taxi trip and fare data-set described here. This notebook may be a good starting point for quickly exploring the code we have provided for Spark 2.0. For a more detailed notebook analyzes the NYC Taxi data, see the next notebook in this list. See the notes following this list that compares these notebooks.
- Spark2.0-pySpark3_NYC_Taxi_Tip_Regression.ipynb: This file shows how to perform data wrangling (Spark SQL and dataframe operations), exploration, modeling and scoring using the NYC Taxi trip and fare data-set described here.
- Spark2.0-pySpark3_Airline_Departure_Delay_Classification.ipynb: This file shows how to perform data wrangling (Spark SQL and dataframe operations), exploration, modeling and scoring using the well-known Airline On-time departure dataset from 2011 and 2012. We integrated the airline dataset with the airport weather data (for example, windspeed, temperature, altitude etc.) prior to modeling, so these weather features can be included in the model.

> **NOTE**
>
> The airline dataset was added to the Spark 2.0 notebooks to better illustrate the use of classification algorithms. See the following links for information about airline on-time departure dataset and weather dataset:
>
> - Airline on-time departure data: https://www.transtats.bts.gov/ONTIME/
> - Airport weather data: https://www.ncdc.noaa.gov/

> **NOTE**
>
> The Spark 2.0 notebooks on the NYC taxi and airline flight delay data-sets can take 10 mins or more to run (depending on the size of your HDI cluster). The first notebook in the above list shows many aspects of the data exploration, visualization and ML model training in a notebook that takes less time to run with down-sampled NYC data set, in which the taxi and fare files have been pre-joined: Spark2.0-pySpark3-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb. This notebook takes a much shorter time to finish (2-3 mins) and may be a good starting point for quickly exploring the code we have provided for Spark 2.0.

For guidance on the operationalization of a Spark 2.0 model and model consumption for scoring, see the Spark 1.6 document on consumption for an example outlining the steps required. To use this example on Spark 2.0, replace the Python code file with this file.

Prerequisites

The following procedures are related to Spark 1.6. For the Spark 2.0 version, use the notebooks described and linked to previously.

1. You must have an Azure subscription. If you do not already have one, see Get Azure free trial.
2. You need a Spark 1.6 cluster to complete this walkthrough. To create one, see the instructions provided in Get started: create Apache Spark on Azure HDInsight. The cluster type and version is specified from the **Select Cluster Type** menu.

> **NOTE**
> For a topic that shows how to use Scala rather than Python to complete tasks for an end-to-end data science process, see the Data Science using Scala with Spark on Azure.

> **WARNING**
> Billing for HDInsight clusters is prorated per minute, whether you use them or not. Be sure to delete your cluster after you finish using it. See how to delete an HDInsight cluster.

The NYC 2013 Taxi data

The NYC Taxi Trip data is about 20 GB of compressed comma-separated values (CSV) files (~48 GB uncompressed), comprising more than 173 million individual trips and the fares paid for each trip. Each trip record includes the pickup and dropoff location and time, anonymized hack (driver's) license number and medallion (taxi's unique id) number. The data covers all trips in the year 2013 and is provided in the following two datasets for each month:

1. The 'trip_data' CSV files contain trip details, such as number of passengers, pick up and dropoff points, trip duration, and trip length. Here are a few sample records:

```
medallion,hack_license,vendor_id,rate_code,store_and_fwd_flag,pickup_datetime,dropoff_datetime,passenger_count,trip_time_in_secs,trip_distance,pickup_longit
```

```
89D227B655E5C82AECF13C3F540D4CF4,BA96DE419E711691B9445D6A6307C170,CMT,1,N,2013-01-01 15:11:48,2013-01-01 15:18:10,4,382,1.00,-73.978165,40.757977,-73.989838,40.751171
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,N,2013-01-06 00:18:35,2013-01-06 00:22:54,1,259,1.50,-74.006683,40.731781,-73.994499,40.75066
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,N,2013-01-05 18:49:41,2013-01-05 18:54:23,1,282,1.10,-74.004707,40.73777,-74.009834,40.726002
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,1,N,2013-01-07 23:54:15,2013-01-07 23:58:20,2,244,.70,-73.974602,40.759945,-73.984734,40.759388
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,1,N,2013-01-07 23:25:03,2013-01-07 23:34:24,1,560,2.10,-73.97625,40.748528,-74.002586,40.747868
```

2. The 'trip_fare' CSV files contain details of the fare paid for each trip, such as payment type, fare amount, surcharge and taxes, tips and tolls, and the total amount paid. Here are a few sample records:

```
medallion, hack_license, vendor_id, pickup_datetime, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount
```

```
89D227B655E5C82AECF13C3F540D4CF4,BA96DE419E711691B9445D6A6307C170,CMT,2013-01-01 15:11:48,CSH,6.5,0,0.5,0,0,7
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,2013-01-06 00:18:35,CSH,6,0.5,0.5,0,0,7
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,2013-01-05 18:49:41,CSH,5.5,1,0.5,0,0,7
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,2013-01-07 23:54:15,CSH,5,0.5,0.5,0,0,6
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,2013-01-07 23:25:03,CSH,9.5,0.5,0.5,0,0,10.5
```

We have taken a 0.1% sample of these files and joined the trip_data and trip_fare CVS files into a single dataset to use as the input dataset for this walkthrough. The unique key to join trip_data and trip_fare is composed of the fields: medallion, hack_licence and pickup_datetime. Each record of the dataset contains the following

attributes representing a NYC Taxi trip:

FIELD	BRIEF DESCRIPTION
medallion	Anonymized taxi medallion (unique taxi id)
hack_license	Anonymized Hackney Carriage License number
vendor_id	Taxi vendor id
rate_code	NYC taxi rate of fare
store_and_fwd_flag	Store and forward flag
pickup_datetime	Pick up date & time
dropoff_datetime	Dropoff date & time
pickup_hour	Pick up hour
pickup_week	Pick up week of the year
weekday	Weekday (range 1-7)
passenger_count	Number of passengers in a taxi trip
trip_time_in_secs	Trip time in seconds
trip_distance	Trip distance traveled in miles
pickup_longitude	Pick up longitude
pickup_latitude	Pick up latitude
dropoff_longitude	Dropoff longitude
dropoff_latitude	Dropoff latitude
direct_distance	Direct distance between pickup and dropoff locations
payment_type	Payment type (cash, credit-card etc.)
fare_amount	Fare amount in
surcharge	Surcharge
mta_tax	MTA Metro Transportation tax
tip_amount	Tip amount
tolls_amount	Tolls amount
total_amount	Total amount
tipped	Tipped (0/1 for no or yes)
tip_class	Tip class (0: $0, 1: $0-5, 2: $6-10, 3: $11-20, 4: > $20)

Execute code from a Jupyter notebook on the Spark cluster

You can launch the Jupyter Notebook from the Azure portal. Find your Spark cluster on your dashboard and click it to enter management page for your cluster. To open the notebook associated with the Spark cluster, click **Cluster Dashboards** -> **Jupyter Notebook**.

You can also browse to `https://CLUSTERNAME.azurehdinsight.net/jupyter` to access the Jupyter Notebooks. Replace the CLUSTERNAME part of this URL with the name of your own cluster. You need the password for your admin account to access the notebooks.

Select PySpark to see a directory that contains a few examples of pre-packaged notebooks that use the PySpark API. The notebooks that contain the code samples for this suite of Spark topic are available at GitHub

You can upload the notebooks directly from GitHub to the Jupyter notebook server on your Spark cluster. On the home page of your Jupyter, click the **Upload** button on the right part of the screen. It opens a file explorer. Here you can paste the GitHub (raw content) URL of the Notebook and click **Open**.

You see the file name on your Jupyter file list with an **Upload** button again. Click this **Upload** button. Now you have imported the notebook. Repeat these steps to upload the other notebooks from this walkthrough.

> **TIP**
> You can right-click the links on your browser and select **Copy Link** to get the GitHub raw content URL. You can paste this URL into the Jupyter Upload file explorer dialog box.

Now you can:

- See the code by clicking the notebook.
- Execute each cell by pressing **SHIFT-ENTER**.
- Run the entire notebook by clicking on **Cell** -> **Run**.
- Use the automatic visualization of queries.

> **TIP**
> The PySpark kernel automatically visualizes the output of SQL (HiveQL) queries. You are given the option to select among several different types of visualizations (Table, Pie, Line, Area, or Bar) by using the **Type** menu buttons in the notebook:

What's next?

Now that you are set up with an HDInsight Spark cluster and have uploaded the Jupyter notebooks, you are ready to work through the topics that correspond to the three PySpark notebooks. They show how to explore your data and then how to create and consume models. The advanced data exploration and modeling notebook shows how to include cross-validation, hyper-parameter sweeping, and model evaluation.

Data Exploration and modeling with Spark: Explore the dataset and create, score, and evaluate the machine learning models by working through the Create binary classification and regression models for data with the Spark MLlib toolkit topic.

Model consumption: To learn how to score the classification and regression models created in this topic, see Score and evaluate Spark-built machine learning models.

Cross-validation and hyperparameter sweeping: See Advanced data exploration and modeling with Spark on how models can be trained using cross-validation and hyper-parameter sweeping

Data Science using Scala and Spark on Azure

11/19/2021 • 34 minutes to read • Edit Online

This article shows you how to use Scala for supervised machine learning tasks with the Spark scalable MLlib and Spark ML packages on an Azure HDInsight Spark cluster. It walks you through the tasks that constitute the Data Science process: data ingestion and exploration, visualization, feature engineering, modeling, and model consumption. The models in the article include logistic and linear regression, random forests, and gradient-boosted trees (GBTs), in addition to two common supervised machine learning tasks:

- Regression problem: Prediction of the tip amount ($) for a taxi trip
- Binary classification: Prediction of tip or no tip (1/0) for a taxi trip

The modeling process requires training and evaluation on a test data set and relevant accuracy metrics. In this article, you can learn how to store these models in Azure Blob storage and how to score and evaluate their predictive performance. This article also covers the more advanced topics of how to optimize models by using cross-validation and hyper-parameter sweeping. The data used is a sample of the 2013 NYC taxi trip and fare data set available on GitHub.

Scala, a language based on the Java virtual machine, integrates object-oriented and functional language concepts. It's a scalable language that is well suited to distributed processing in the cloud, and runs on Azure Spark clusters.

Spark is an open-source parallel-processing framework that supports in-memory processing to boost the performance of big data analytics applications. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory distributed computation capabilities make it a good choice for iterative algorithms in machine learning and graph computations. The spark.ml package provides a uniform set of high-level APIs built on top of data frames that can help you create and tune practical machine learning pipelines. MLlib is Spark's scalable machine learning library, which brings modeling capabilities to this distributed environment.

HDInsight Spark is the Azure-hosted offering of open-source Spark. It also includes support for Jupyter Scala notebooks on the Spark cluster, and can run Spark SQL interactive queries to transform, filter, and visualize data stored in Azure Blob storage. The Scala code snippets in this article that provide the solutions and show the relevant plots to visualize the data run in Jupyter notebooks installed on the Spark clusters. The modeling steps in these topics have code that shows you how to train, evaluate, save, and consume each type of model.

The setup steps and code in this article are for Azure HDInsight 3.4 Spark 1.6. However, the code in this article and in the Scala Jupyter Notebook are generic and should work on any Spark cluster. The cluster setup and management steps might be slightly different from what is shown in this article if you are not using HDInsight Spark.

> **NOTE**
>
> For a topic that shows you how to use Python rather than Scala to complete tasks for an end-to-end Data Science process, see Data Science using Spark on Azure HDInsight.

Prerequisites

- You must have an Azure subscription. If you do not already have one, get an Azure free trial.
- You need an Azure HDInsight 3.4 Spark 1.6 cluster to complete the following procedures. To create a cluster,

see the instructions in Get started: Create Apache Spark on Azure HDInsight. Set the cluster type and version on the **Select Cluster Type** menu.

> **WARNING**
>
> Billing for HDInsight clusters is prorated per minute, whether you use them or not. Be sure to delete your cluster after you finish using it. See how to delete an HDInsight cluster.

For a description of the NYC taxi trip data and instructions on how to execute code from a Jupyter notebook on the Spark cluster, see the relevant sections in Overview of Data Science using Spark on Azure HDInsight.

Execute Scala code from a Jupyter notebook on the Spark cluster

You can launch a Jupyter notebook from the Azure portal. Find the Spark cluster on your dashboard, and then click it to enter the management page for your cluster. Next, click **Cluster Dashboards**, and then click **Jupyter Notebook** to open the notebook associated with the Spark cluster.

You also can access Jupyter notebooks at https://<clustername>.azurehdinsight.net/jupyter. Replace *clustername* with the name of your cluster. You need the password for your administrator account to access the Jupyter notebooks.

Select **Scala** to see a directory that has a few examples of prepackaged notebooks that use the PySpark API. The Exploration Modeling and Scoring using Scala.ipynb notebook that contains the code samples for this suite of Spark topics is available on GitHub.

You can upload the notebook directly from GitHub to the Jupyter Notebook server on your Spark cluster. On your Jupyter home page, click the **Upload** button. In the file explorer, paste the GitHub (raw content) URL of the Scala notebook, and then click **Open**. The Scala notebook is available at the following URL:

Exploration-Modeling-and-Scoring-using-Scala.ipynb

Setup: Preset Spark and Hive contexts, Spark magics, and Spark

libraries

Preset Spark and Hive contexts

```
# SET THE START TIME
import java.util.Calendar
val beginningTime = Calendar.getInstance().getTime()
```

The Spark kernels that are provided with Jupyter notebooks have preset contexts. You don't need to explicitly set the Spark or Hive contexts before you start working with the application you are developing. The preset contexts are:

- `sc` for SparkContext
- `sqlContext` for HiveContext

Spark magics

The Spark kernel provides some predefined "magics," which are special commands that you can call with `%%`. Two of these commands are used in the following code samples.

- `%%local` specifies that the code in subsequent lines will be executed locally. The code must be valid Scala code.
- `%%sql -o <variable name>` executes a Hive query against `sqlContext`. If the `-o` parameter is passed, the result of the query is persisted in the `%%local` Scala context as a Spark data frame.

For more information about the kernels for Jupyter notebooks and their predefined "magics" that you call with `%%` (for example, `%%local`), see Kernels available for Jupyter notebooks with HDInsight Spark Linux clusters on HDInsight.

Import libraries

Import the Spark, MLlib, and other libraries you'll need by using the following code.

```
# IMPORT SPARK AND JAVA LIBRARIES
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.functions._
import java.text.SimpleDateFormat
import java.util.Calendar
import sqlContext.implicits._
import org.apache.spark.sql.Row

# IMPORT SPARK SQL FUNCTIONS
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType, FloatType, DoubleType}
import org.apache.spark.sql.functions.rand

# IMPORT SPARK ML FUNCTIONS
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler, OneHotEncoder, VectorIndexer, Binarizer}
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit, CrossValidator}
import org.apache.spark.ml.regression.{LinearRegression, LinearRegressionModel, RandomForestRegressor,
RandomForestRegressionModel, GBTRegressor, GBTRegressionModel}
import org.apache.spark.ml.classification.{LogisticRegression, LogisticRegressionModel,
RandomForestClassifier, RandomForestClassificationModel, GBTClassifier, GBTClassificationModel}
import org.apache.spark.ml.evaluation.{BinaryClassificationEvaluator, RegressionEvaluator,
MulticlassClassificationEvaluator}

# IMPORT SPARK MLLIB FUNCTIONS
import org.apache.spark.mllib.linalg.{Vector, Vectors}
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.classification.{LogisticRegressionWithLBFGS, LogisticRegressionModel}
import org.apache.spark.mllib.regression.{LabeledPoint, LinearRegressionWithSGD, LinearRegressionModel}
import org.apache.spark.mllib.tree.{GradientBoostedTrees, RandomForest}
import org.apache.spark.mllib.tree.configuration.BoostingStrategy
import org.apache.spark.mllib.tree.model.{GradientBoostedTreesModel, RandomForestModel, Predict}
import org.apache.spark.mllib.evaluation.{BinaryClassificationMetrics, MulticlassMetrics, RegressionMetrics}

# SPECIFY SQLCONTEXT
val sqlContext = new SQLContext(sc)
```

Data ingestion

The first step in the Data Science process is to ingest the data that you want to analyze. You bring the data from external sources or systems where it resides into your data exploration and modeling environment. In this article, the data you ingest is a joined 0.1% sample of the taxi trip and fare file (stored as a .tsv file). The data exploration and modeling environment is Spark. This section contains the code to complete the following series of tasks:

1. Set directory paths for data and model storage.
2. Read in the input data set (stored as a .tsv file).
3. Define a schema for the data and clean the data.
4. Create a cleaned data frame and cache it in memory.
5. Register the data as a temporary table in SQLContext.
6. Query the table and import the results into a data frame.

Set directory paths for storage locations in Azure Blob storage

Spark can read and write to Azure Blob storage. You can use Spark to process any of your existing data, and then store the results again in Blob storage.

To save models or files in Blob storage, you need to properly specify the path. Reference the default container attached to the Spark cluster by using a path that begins with `wasb:///`. Reference other locations by using `wasb://`.

The following code sample specifies the location of the input data to be read and the path to Blob storage that is

attached to the Spark cluster where the model will be saved.

```
# SET PATHS TO DATA AND MODEL FILE LOCATIONS
# INGEST DATA AND SPECIFY HEADERS FOR COLUMNS
val taxi_train_file =
sc.textFile("wasb://mllibwalkthroughs@cdspsparksamples.blob.core.windows.net/Data/NYCTaxi/JoinedTaxiTripFare
.Point1Pct.Train.tsv")
val header = taxi_train_file.first;

# SET THE MODEL STORAGE DIRECTORY PATH
# NOTE THAT THE FINAL BACKSLASH IN THE PATH IS REQUIRED.
val modelDir = "wasb:///user/remoteuser/NYCTaxi/Models/";
```

Import data, create an RDD, and define a data frame according to the schema

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# DEFINE THE SCHEMA BASED ON THE HEADER OF THE FILE
val sqlContext = new SQLContext(sc)
val taxi_schema = StructType(
    Array(
        StructField("medallion", StringType, true),
        StructField("hack_license", StringType, true),
        StructField("vendor_id", StringType, true),
        StructField("rate_code", DoubleType, true),
        StructField("store_and_fwd_flag", StringType, true),
        StructField("pickup_datetime", StringType, true),
        StructField("dropoff_datetime", StringType, true),
        StructField("pickup_hour", DoubleType, true),
        StructField("pickup_week", DoubleType, true),
        StructField("weekday", DoubleType, true),
        StructField("passenger_count", DoubleType, true),
        StructField("trip_time_in_secs", DoubleType, true),
        StructField("trip_distance", DoubleType, true),
        StructField("pickup_longitude", DoubleType, true),
        StructField("pickup_latitude", DoubleType, true),
        StructField("dropoff_longitude", DoubleType, true),
        StructField("dropoff_latitude", DoubleType, true),
        StructField("direct_distance", StringType, true),
        StructField("payment_type", StringType, true),
        StructField("fare_amount", DoubleType, true),
        StructField("surcharge", DoubleType, true),
        StructField("mta_tax", DoubleType, true),
        StructField("tip_amount", DoubleType, true),
        StructField("tolls_amount", DoubleType, true),
        StructField("total_amount", DoubleType, true),
        StructField("tipped", DoubleType, true),
        StructField("tip_class", DoubleType, true)
        )
    )

# CAST VARIABLES ACCORDING TO THE SCHEMA
val taxi_temp = (taxi_train_file.map(_.split("\t"))
                        .filter((r) => r(0) != "medallion")
                         .map(p => Row(p(0), p(1), p(2),
                            p(3).toDouble, p(4), p(5), p(6), p(7).toDouble, p(8).toDouble, p(9).toDouble,
p(10).toDouble,
                            p(11).toDouble, p(12).toDouble, p(13).toDouble, p(14).toDouble, p(15).toDouble,
p(16).toDouble,
                            p(17), p(18), p(19).toDouble, p(20).toDouble, p(21).toDouble, p(22).toDouble,
                            p(23).toDouble, p(24).toDouble, p(25).toDouble, p(26).toDouble)))

# CREATE AN INITIAL DATA FRAME AND DROP COLUMNS, AND THEN CREATE A CLEANED DATA FRAME BY FILTERING FOR
UNWANTED VALUES OR OUTLIERS
val taxi_train_df = sqlContext.createDataFrame(taxi_temp, taxi_schema)
```

```
val taxi_df_train_cleaned = (taxi_train_df.drop(taxi_train_df.col("medallion"))
        .drop(taxi_train_df.col("hack_license")).drop(taxi_train_df.col("store_and_fwd_flag"))
        .drop(taxi_train_df.col("pickup_datetime")).drop(taxi_train_df.col("dropoff_datetime"))
        .drop(taxi_train_df.col("pickup_longitude")).drop(taxi_train_df.col("pickup_latitude"))
        .drop(taxi_train_df.col("dropoff_longitude")).drop(taxi_train_df.col("dropoff_latitude"))
        .drop(taxi_train_df.col("surcharge")).drop(taxi_train_df.col("mta_tax"))
        .drop(taxi_train_df.col("direct_distance")).drop(taxi_train_df.col("tolls_amount"))
        .drop(taxi_train_df.col("total_amount")).drop(taxi_train_df.col("tip_class"))
        .filter("passenger_count > 0 and passenger_count < 8 AND payment_type in ('CSH', 'CRD') AND
tip_amount >= 0 AND tip_amount < 30 AND fare_amount >= 1 AND fare_amount < 150 AND trip_distance > 0 AND
trip_distance < 100 AND trip_time_in_secs > 30 AND trip_time_in_secs < 7200"));

# CACHE AND MATERIALIZE THE CLEANED DATA FRAME IN MEMORY
taxi_df_train_cleaned.cache()
taxi_df_train_cleaned.count()

# REGISTER THE DATA FRAME AS A TEMPORARY TABLE IN SQLCONTEXT
taxi_df_train_cleaned.registerTempTable("taxi_train")

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");
```

Output:

Time to run the cell: 8 seconds.

Query the table and import results in a data frame

Next, query the table for fare, passenger, and tip data; filter out corrupt and outlying data; and print several rows.

```
# QUERY THE DATA
val sqlStatement = """
    SELECT fare_amount, passenger_count, tip_amount, tipped
    FROM taxi_train
    WHERE passenger_count > 0 AND passenger_count < 7
    AND fare_amount > 0 AND fare_amount < 200
    AND payment_type in ('CSH', 'CRD')
    AND tip_amount > 0 AND tip_amount < 25
"""
val sqlResultsDF = sqlContext.sql(sqlStatement)

# SHOW ONLY THE TOP THREE ROWS
sqlResultsDF.show(3)
```

Output:

FARE_AMOUNT	PASSENGER_COUNT	TIP_AMOUNT	TIPPED
13.5	1.0	2.9	1.0
16.0	2.0	3.4	1.0
10.5	2.0	1.0	1.0

Data exploration and visualization

After you bring the data into Spark, the next step in the Data Science process is to gain a deeper understanding of the data through exploration and visualization. In this section, you examine the taxi data by using SQL queries. Then, import the results into a data frame to plot the target variables and prospective features for visual

inspection by using the automatic visualization Jupyter feature.

Use local and SQL magic to plot data

By default, the output of any code snippet that you run from a Jupyter notebook is available within the context of the session that is persisted on the worker nodes. If you want to save a trip to the worker nodes for every computation, and if all the data that you need for your computation is available locally on the Jupyter server node (which is the head node), you can use the `%%local` magic to run the code snippet on the Jupyter server.

- **SQL magic** (`%%sql`). The HDInsight Spark kernel supports easy inline HiveQL queries against SQLContext. The (`-o VARIABLE_NAME`) argument persists the output of the SQL query as a Pandas data frame on the Jupyter server. This setting means the output will be available in the local mode.
- `%%local` **magic**. The `%%local` magic runs the code locally on the Jupyter server, which is the head node of the HDInsight cluster. Typically, you use `%%local` magic in conjunction with the `%%sql` magic with the `-o` parameter. The `-o` parameter would persist the output of the SQL query locally, and then `%%local` magic would trigger the next set of code snippet to run locally against the output of the SQL queries that is persisted locally.

Query the data by using SQL

This query retrieves the taxi trips by fare amount, passenger count, and tip amount.

```
# RUN THE SQL QUERY
%%sql -q -o sqlResults
SELECT fare_amount, passenger_count, tip_amount, tipped FROM taxi_train WHERE passenger_count > 0 AND
passenger_count < 7 AND fare_amount > 0 AND fare_amount < 200 AND payment_type in ('CSH', 'CRD') AND
tip_amount > 0 AND tip_amount < 25
```

In the following code, the `%%local` magic creates a local data frame, sqlResults. You can use sqlResults to plot by using matplotlib.

> **TIP**
> Local magic is used multiple times in this article. If your data set is large, please sample to create a data frame that can fit in local memory.

Plot the data

You can plot by using Python code after the data frame is in local context as a Pandas data frame.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER
%%local

# USE THE JUPYTER AUTO-PLOTTING FEATURE TO CREATE INTERACTIVE FIGURES.
# CLICK THE TYPE OF PLOT TO GENERATE (LINE, AREA, BAR, ETC.)
sqlResults
```

The Spark kernel automatically visualizes the output of SQL (HiveQL) queries after you run the code. You can choose between several types of visualizations:

- Table
- Pie
- Line
- Area
- Bar

Here's the code to plot the data:

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
import matplotlib.pyplot as plt
%matplotlib inline

# PLOT TIP BY PAYMENT TYPE AND PASSENGER COUNT
ax1 = sqlResults[['tip_amount']].plot(kind='hist', bins=25, facecolor='lightblue')
ax1.set_title('Tip amount distribution')
ax1.set_xlabel('Tip Amount ($)')
ax1.set_ylabel('Counts')
plt.suptitle('')
plt.show()

# PLOT TIP BY PASSENGER COUNT
ax2 = sqlResults.boxplot(column=['tip_amount'], by=['passenger_count'])
ax2.set_title('Tip amount by Passenger count')
ax2.set_xlabel('Passenger count')
ax2.set_ylabel('Tip Amount ($)')
plt.suptitle('')
plt.show()

# PLOT TIP AMOUNT BY FARE AMOUNT; SCALE POINTS BY PASSENGER COUNT
ax = sqlResults.plot(kind='scatter', x= 'fare_amount', y = 'tip_amount', c='blue', alpha = 0.10, s=5*
(sqlResults.passenger_count))
ax.set_title('Tip amount by Fare amount')
ax.set_xlabel('Fare Amount ($)')
ax.set_ylabel('Tip Amount ($)')
plt.axis([-2, 80, -2, 20])
plt.show()
```

Output:

Create features and transform features, and then prep data for input into modeling functions

For tree-based modeling functions from Spark ML and MLlib, you have to prepare target and features by using a variety of techniques, such as binning, indexing, one-hot encoding, and vectorization. Here are the procedures to follow in this section:

1. Create a new feature by **binning** hours into traffic time buckets.
2. Apply **indexing and one-hot encoding** to categorical features.
3. **Sample and split the data set** into training and test fractions.
4. **Specify training variable and features**, and then create indexed or one-hot encoded training and testing input labeled point resilient distributed datasets (RDDs) or data frames.
5. Automatically **categorize and vectorize features and targets** to use as inputs for machine learning models.

Create a new feature by binning hours into traffic time buckets

This code shows you how to create a new feature by binning hours into traffic time buckets and how to cache the resulting data frame in memory. Where RDDs and data frames are used repeatedly, caching leads to improved execution times. Accordingly, you'll cache RDDs and data frames at several stages in the following procedures.

```
# CREATE FOUR BUCKETS FOR TRAFFIC TIMES
val sqlStatement = """
    SELECT *,
    CASE
     WHEN (pickup_hour <= 6 OR pickup_hour >= 20) THEN "Night"
     WHEN (pickup_hour >= 7 AND pickup_hour <= 10) THEN "AMRush"
     WHEN (pickup_hour >= 11 AND pickup_hour <= 15) THEN "Afternoon"
     WHEN (pickup_hour >= 16 AND pickup_hour <= 19) THEN "PMRush"
    END as TrafficTimeBins
    FROM taxi_train
"""
val taxi_df_train_with_newFeatures = sqlContext.sql(sqlStatement)

# CACHE THE DATA FRAME IN MEMORY AND MATERIALIZE THE DATA FRAME IN MEMORY
taxi_df_train_with_newFeatures.cache()
taxi_df_train_with_newFeatures.count()
```

Indexing and one-hot encoding of categorical features

The modeling and predict functions of MLlib require features with categorical input data to be indexed or encoded prior to use. This section shows you how to index or encode categorical features for input into the modeling functions.

You need to index or encode your models in different ways, depending on the model. For example, logistic and linear regression models require one-hot encoding. For example, a feature with three categories can be expanded into three feature columns. Each column would contain 0 or 1 depending on the category of an observation. MLlib provides the OneHotEncoder function for one-hot encoding. This encoder maps a column of label indices to a column of binary vectors with at most a single one-value. With this encoding, algorithms that expect numerical valued features, such as logistic regression, can be applied to categorical features.

Here you transform only four variables to show examples, which are character strings. You also can index other variables, such as weekday, represented by numerical values, as categorical variables.

For indexing, use `StringIndexer()`, and for one-hot encoding, use `OneHotEncoder()` functions from MLlib. Here is the code to index and encode categorical features:

```
# CREATE INDEXES AND ONE-HOT ENCODED VECTORS FOR SEVERAL CATEGORICAL FEATURES

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# INDEX AND ENCODE VENDOR_ID
val stringIndexer = new
StringIndexer().setInputCol("vendor_id").setOutputCol("vendorIndex").fit(taxi_df_train_with_newFeatures)
val indexed = stringIndexer.transform(taxi_df_train_with_newFeatures)
val encoder = new OneHotEncoder().setInputCol("vendorIndex").setOutputCol("vendorVec")
val encoded1 = encoder.transform(indexed)

# INDEX AND ENCODE RATE_CODE
val stringIndexer = new StringIndexer().setInputCol("rate_code").setOutputCol("rateIndex").fit(encoded1)
val indexed = stringIndexer.transform(encoded1)
val encoder = new OneHotEncoder().setInputCol("rateIndex").setOutputCol("rateVec")
val encoded2 = encoder.transform(indexed)

# INDEX AND ENCODE PAYMENT_TYPE
val stringIndexer = new
StringIndexer().setInputCol("payment_type").setOutputCol("paymentIndex").fit(encoded2)
val indexed = stringIndexer.transform(encoded2)
val encoder = new OneHotEncoder().setInputCol("paymentIndex").setOutputCol("paymentVec")
val encoded3 = encoder.transform(indexed)

# INDEX AND TRAFFIC TIME BINS
val stringIndexer = new
StringIndexer().setInputCol("TrafficTimeBins").setOutputCol("TrafficTimeBinsIndex").fit(encoded3)
val indexed = stringIndexer.transform(encoded3)
val encoder = new OneHotEncoder().setInputCol("TrafficTimeBinsIndex").setOutputCol("TrafficTimeBinsVec")
val encodedFinal = encoder.transform(indexed)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");
```

Output:

Time to run the cell: 4 seconds.

Sample and split the data set into training and test fractions

This code creates a random sampling of the data (25%, in this example). Although sampling is not required for this example due to the size of the data set, the article shows you how you can sample so that you know how to use it for your own problems when needed. When samples are large, this can save significant time while you train models. Next, split the sample into a training part (75%, in this example) and a testing part (25%, in this example) to use in classification and regression modeling.

Add a random number (between 0 and 1) to each row (in a "rand" column) that can be used to select cross-

validation folds during training.

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# SPECIFY SAMPLING AND SPLITTING FRACTIONS
val samplingFraction = 0.25;
val trainingFraction = 0.75;
val testingFraction = (1-trainingFraction);
val seed = 1234;
val encodedFinalSampledTmp = encodedFinal.sample(withReplacement = false, fraction = samplingFraction, seed
= seed)
val sampledDFcount = encodedFinalSampledTmp.count().toInt

val generateRandomDouble = udf(() => {
    scala.util.Random.nextDouble
})

# ADD A RANDOM NUMBER FOR CROSS-VALIDATION
val encodedFinalSampled = encodedFinalSampledTmp.withColumn("rand", generateRandomDouble());

# SPLIT THE SAMPLED DATA FRAME INTO TRAIN AND TEST, WITH A RANDOM COLUMN ADDED FOR DOING CROSS-VALIDATION
(SHOWN LATER)
# INCLUDE A RANDOM COLUMN FOR CREATING CROSS-VALIDATION FOLDS
val splits = encodedFinalSampled.randomSplit(Array(trainingFraction, testingFraction), seed = seed)
val trainData = splits(0)
val testData = splits(1)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");
```

Output:

Time to run the cell: 2 seconds.

Specify training variable and features, and then create indexed or one-hot encoded training and testing input labeled point RDDs or data frames

This section contains code that shows you how to index categorical text data as a labeled point data type, and encode it so you can use it to train and test MLlib logistic regression and other classification models. Labeled point objects are RDDs that are formatted in a way that is needed as input data by most of machine learning algorithms in MLlib. A labeled point is a local vector, either dense or sparse, associated with a label/response.

In this code, you specify the target (dependent) variable and the features to use to train models. Then, you create indexed or one-hot encoded training and testing input labeled point RDDs or data frames.

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# MAP NAMES OF FEATURES AND TARGETS FOR CLASSIFICATION AND REGRESSION PROBLEMS
val featuresIndOneHot = List("paymentVec", "vendorVec", "rateVec", "TrafficTimeBinsVec", "pickup_hour",
"weekday", "passenger_count", "trip_time_in_secs", "trip_distance",
"fare_amount").map(encodedFinalSampled.columns.indexOf(_))
val featuresIndIndex = List("paymentIndex", "vendorIndex", "rateIndex", "TrafficTimeBinsIndex",
"pickup_hour", "weekday", "passenger_count", "trip_time_in_secs", "trip_distance",
"fare_amount").map(encodedFinalSampled.columns.indexOf(_))

# SPECIFY THE TARGET FOR CLASSIFICATION ('tipped') AND REGRESSION ('tip_amount') PROBLEMS
val targetIndBinary = List("tipped").map(encodedFinalSampled.columns.indexOf(_))
val targetIndRegression = List("tip_amount").map(encodedFinalSampled.columns.indexOf(_))

# CREATE INDEXED LABELED POINT RDD OBJECTS
val indexedTRAINbinary = trainData.rdd.map(r => LabeledPoint(r.getDouble(targetIndBinary(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))
val indexedTESTbinary = testData.rdd.map(r => LabeledPoint(r.getDouble(targetIndBinary(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))
val indexedTRAINreg = trainData.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))
val indexedTESTreg = testData.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)))

# CREATE INDEXED DATA FRAMES THAT YOU CAN USE TO TRAIN BY USING SPARK ML FUNCTIONS
val indexedTRAINbinaryDF = indexedTRAINbinary.toDF()
val indexedTESTbinaryDF = indexedTESTbinary.toDF()
val indexedTRAINregDF = indexedTRAINreg.toDF()
val indexedTESTregDF = indexedTESTreg.toDF()

# CREATE ONE-HOT ENCODED (VECTORIZED) DATA FRAMES THAT YOU CAN USE TO TRAIN BY USING SPARK ML FUNCTIONS
val assemblerOneHot = new VectorAssembler().setInputCols(Array("paymentVec", "vendorVec", "rateVec",
"TrafficTimeBinsVec", "pickup_hour", "weekday", "passenger_count", "trip_time_in_secs", "trip_distance",
"fare_amount")).setOutputCol("features")
val OneHotTRAIN = assemblerOneHot.transform(trainData)
val OneHotTEST = assemblerOneHot.transform(testData)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");
```

Output:

Time to run the cell: 4 seconds.

Automatically categorize and vectorize features and targets to use as inputs for machine learning models

Use Spark ML to categorize the target and features to use in tree-based modeling functions. The code completes two tasks:

- Creates a binary target for classification by assigning a value of 0 or 1 to each data point between 0 and 1 by using a threshold value of 0.5.
- Automatically categorizes features. If the number of distinct numerical values for any feature is less than 32, that feature is categorized.

Here's the code for these two tasks.

```
# CATEGORIZE FEATURES AND BINARIZE THE TARGET FOR THE BINARY CLASSIFICATION PROBLEM

# TRAIN DATA
val indexer = new VectorIndexer().setInputCol("features").setOutputCol("featuresCat").setMaxCategories(32)
val indexerModel = indexer.fit(indexedTRAINbinaryDF)
val indexedTrainwithCatFeat = indexerModel.transform(indexedTRAINbinaryDF)
val binarizer: Binarizer = new Binarizer().setInputCol("label").setOutputCol("labelBin").setThreshold(0.5)
val indexedTRAINwithCatFeatBinTarget = binarizer.transform(indexedTrainwithCatFeat)

# TEST DATA
val indexerModel = indexer.fit(indexedTESTbinaryDF)
val indexedTrainwithCatFeat = indexerModel.transform(indexedTESTbinaryDF)
val binarizer: Binarizer = new Binarizer().setInputCol("label").setOutputCol("labelBin").setThreshold(0.5)
val indexedTESTwithCatFeatBinTarget = binarizer.transform(indexedTrainwithCatFeat)

# CATEGORIZE FEATURES FOR THE REGRESSION PROBLEM
# CREATE PROPERLY INDEXED AND CATEGORIZED DATA FRAMES FOR TREE-BASED MODELS

# TRAIN DATA
val indexer = new VectorIndexer().setInputCol("features").setOutputCol("featuresCat").setMaxCategories(32)
val indexerModel = indexer.fit(indexedTRAINregDF)
val indexedTRAINwithCatFeat = indexerModel.transform(indexedTRAINregDF)

# TEST DATA
val indexerModel = indexer.fit(indexedTESTbinaryDF)
val indexedTESTwithCatFeat = indexerModel.transform(indexedTESTregDF)
```

Binary classification model: Predict whether a tip should be paid

In this section, you create three types of binary classification models to predict whether or not a tip should be paid:

- A **logistic regression model** by using the Spark ML `LogisticRegression()` function
- A **random forest classification model** by using the Spark ML `RandomForestClassifier()` function
- A **gradient boosting tree classification model** by using the MLlib `GradientBoostedTrees()` function

Create a logistic regression model

Next, create a logistic regression model by using the Spark ML `LogisticRegression()` function. You create the model building code in a series of steps:

1. **Train the model** data with one parameter set.
2. **Evaluate the model** on a test data set with metrics.
3. **Save the model** in Blob storage for future consumption.
4. **Score the model** against test data.
5. **Plot the results** with receiver operating characteristic (ROC) curves.

Here's the code for these procedures:

```
# CREATE A LOGISTIC REGRESSION MODEL
val lr = new
LogisticRegression().setLabelCol("tipped").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3).setEla
sticNetParam(0.8)
val lrModel = lr.fit(OneHotTRAIN)

# PREDICT ON THE TEST DATA SET
val predictions = lrModel.transform(OneHotTEST)

# SELECT `BinaryClassificationEvaluator()` TO COMPUTE THE TEST ERROR
val evaluator = new
BinaryClassificationEvaluator().setLabelCol("tipped").setRawPredictionCol("probability").setMetricName("area
UnderROC")
val ROC = evaluator.evaluate(predictions)
println("ROC on test data = " + ROC)

# SAVE THE MODEL
val datestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "LogisticRegression__"
val filename = modelDir.concat(modelName).concat(datestamp)
lrModel.save(filename);
```

Load, score, and save the results.

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# LOAD THE SAVED MODEL AND SCORE THE TEST DATA SET
val savedModel = org.apache.spark.ml.classification.LogisticRegressionModel.load(filename)
println(s"Coefficients: ${savedModel.coefficients} Intercept: ${savedModel.intercept}")

# SCORE THE MODEL ON THE TEST DATA
val predictions = savedModel.transform(OneHotTEST).select("tipped","probability","rawPrediction")
predictions.registerTempTable("testResults")

# SELECT `BinaryClassificationEvaluator()` TO COMPUTE THE TEST ERROR
val evaluator = new
BinaryClassificationEvaluator().setLabelCol("tipped").setRawPredictionCol("probability").setMetricName("area
UnderROC")
val ROC = evaluator.evaluate(predictions)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# PRINT THE ROC RESULTS
println("ROC on test data = " + ROC)
```

Output:

ROC on test data = 0.9827381497557599

Use Python on local Pandas data frames to plot the ROC curve.

```
# QUERY THE RESULTS
%%sql -q -o sqlResults
SELECT tipped, probability from testResults

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
%matplotlib inline
from sklearn.metrics import roc_curve,auc

sqlResults['probFloat'] = sqlResults.apply(lambda row: row['probability'].values()[0][1], axis=1)
predictions_pddf = sqlResults[["tipped","probFloat"]]

# PREDICT THE ROC CURVE
# predictions_pddf = sqlResults.rename(columns={'_1': 'probability', 'tipped': 'label'})
prob = predictions_pddf["probFloat"]
fpr, tpr, thresholds = roc_curve(predictions_pddf['tipped'], prob, pos_label=1);
roc_auc = auc(fpr, tpr)

# PLOT THE ROC CURVE
plt.figure(figsize=(5,5))
plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], 'k--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc="lower right")
plt.show()
```

Output:

Create a random forest classification model

Next, create a random forest classification model by using the Spark ML `RandomForestClassifier()` function, and then evaluate the model on test data.

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# CREATE THE RANDOM FOREST CLASSIFIER MODEL
val rf = new
RandomForestClassifier().setLabelCol("labelBin").setFeaturesCol("featuresCat").setNumTrees(10).setSeed(1234)

# FIT THE MODEL
val rfModel = rf.fit(indexedTRAINwithCatFeatBinTarget)
val predictions = rfModel.transform(indexedTESTwithCatFeatBinTarget)

# EVALUATE THE MODEL
val evaluator = new
MulticlassClassificationEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("f1")
val Test_f1Score = evaluator.evaluate(predictions)
println("F1 score on test data: " + Test_f1Score);

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# CALCULATE BINARY CLASSIFICATION EVALUATION METRICS
val evaluator = new
BinaryClassificationEvaluator().setLabelCol("label").setRawPredictionCol("probability").setMetricName("areaU
nderROC")
val ROC = evaluator.evaluate(predictions)
println("ROC on test data = " + ROC)
```

Output:

ROC on test data = 0.9847103571552683

Create a GBT classification model

Next, create a GBT classification model by using MLlib's `GradientBoostedTrees()` function, and then evaluate the model on test data.

```
# TRAIN A GBT CLASSIFICATION MODEL BY USING MLLIB AND A LABELED POINT

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# DEFINE THE GBT CLASSIFICATION MODEL
val boostingStrategy = BoostingStrategy.defaultParams("Classification")
boostingStrategy.numIterations = 20
boostingStrategy.treeStrategy.numClasses = 2
boostingStrategy.treeStrategy.maxDepth = 5
boostingStrategy.treeStrategy.categoricalFeaturesInfo = Map[Int, Int]((0,2),(1,2),(2,6),(3,4))

# TRAIN THE MODEL
val gbtModel = GradientBoostedTrees.train(indexedTRAINbinary, boostingStrategy)

# SAVE THE MODEL IN BLOB STORAGE
val datestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "GBT_Classification__"
val filename = modelDir.concat(modelName).concat(datestamp)
gbtModel.save(sc, filename);

# EVALUATE THE MODEL ON TEST INSTANCES AND THE COMPUTE TEST ERROR
val labelAndPreds = indexedTESTbinary.map { point =>
  val prediction = gbtModel.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / indexedTRAINbinary.count()
//println("Learned classification GBT model:\n" + gbtModel.toDebugString)
println("Test Error = " + testErr)

# USE BINARY AND MULTICLASS METRICS TO EVALUATE THE MODEL ON THE TEST DATA
val metrics = new MulticlassMetrics(labelAndPreds)
println(s"Precision: ${metrics.precision}")
println(s"Recall: ${metrics.recall}")
println(s"F1 Score: ${metrics.fMeasure}")

val metrics = new BinaryClassificationMetrics(labelAndPreds)
println(s"Area under PR curve: ${metrics.areaUnderPR}")
println(s"Area under ROC curve: ${metrics.areaUnderROC}")

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# PRINT THE ROC METRIC
println(s"Area under ROC curve: ${metrics.areaUnderROC}")
```

Output:

Area under ROC curve: 0.9846895479241554

Regression model: Predict tip amount

In this section, you create two types of regression models to predict the tip amount:

- A **regularized linear regression model** by using the Spark ML `LinearRegression()` function. You'll save the model and evaluate the model on test data.
- A **gradient-boosting tree regression model** by using the Spark ML `GBTRegressor()` function.

Create a regularized linear regression model

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# CREATE A REGULARIZED LINEAR REGRESSION MODEL BY USING THE SPARK ML FUNCTION AND DATA FRAMES
val lr = new
LinearRegression().setLabelCol("tip_amount").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3).setE
lasticNetParam(0.8)

# FIT THE MODEL BY USING DATA FRAMES
val lrModel = lr.fit(OneHotTRAIN)
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

# SUMMARIZE THE MODEL OVER THE TRAINING SET AND PRINT METRICS
val trainingSummary = lrModel.summary
println(s"numIterations: ${trainingSummary.totalIterations}")
println(s"objectiveHistory: ${trainingSummary.objectiveHistory.toList}")
trainingSummary.residuals.show()
println(s"RMSE: ${trainingSummary.rootMeanSquaredError}")
println(s"r2: ${trainingSummary.r2}")

# SAVE THE MODEL IN AZURE BLOB STORAGE
val datestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "LinearRegression__"
val filename = modelDir.concat(modelName).concat(datestamp)
lrModel.save(filename);

# PRINT THE COEFFICIENTS
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

# SCORE THE MODEL ON TEST DATA
val predictions = lrModel.transform(OneHotTEST)

# EVALUATE THE MODEL ON TEST DATA
val evaluator = new
RegressionEvaluator().setLabelCol("tip_amount").setPredictionCol("prediction").setMetricName("r2")
val r2 = evaluator.evaluate(predictions)
println("R-sqr on test data = " + r2)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");
```

Output:

Time to run the cell: 13 seconds.

```
# LOAD A SAVED LINEAR REGRESSION MODEL FROM BLOB STORAGE AND SCORE A TEST DATA SET

# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# LOAD A SAVED LINEAR REGRESSION MODEL FROM AZURE BLOB STORAGE
val savedModel = org.apache.spark.ml.regression.LinearRegressionModel.load(filename)
println(s"Coefficients: ${savedModel.coefficients} Intercept: ${savedModel.intercept}")

# SCORE THE MODEL ON TEST DATA
val predictions = savedModel.transform(OneHotTEST).select("tip_amount","prediction")
predictions.registerTempTable("testResults")

# EVALUATE THE MODEL ON TEST DATA
val evaluator = new
RegressionEvaluator().setLabelCol("tip_amount").setPredictionCol("prediction").setMetricName("r2")
val r2 = evaluator.evaluate(predictions)
println("R-sqr on test data = " + r2)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# PRINT THE RESULTS
println("R-sqr on test data = " + r2)
```

Output:

R-sqr on test data = 0.5960320470835743

Next, query the test results as a data frame and use AutoVizWidget and matplotlib to visualize it.

```
# RUN A SQL QUERY
%%sql -q -o sqlResults
select * from testResults

# RUN THE CODE LOCALLY ON THE JUPYTER SERVER
%%local

# USE THE JUPYTER AUTO-PLOTTING FEATURE TO CREATE INTERACTIVE FIGURES
# CLICK THE TYPE OF PLOT TO GENERATE (LINE, AREA, BAR, AND SO ON)
sqlResults
```

The code creates a local data frame from the query output and plots the data. The `%%local` magic creates a local data frame, `sqlResults`, which you can use to plot with matplotlib.

> **NOTE**
>
> This Spark magic is used multiple times in this article. If the amount of data is large, you should sample to create a data frame that can fit in local memory.

Create plots by using Python matplotlib.

```
# RUN THE CODE LOCALLY ON THE JUPYTER SERVER AND IMPORT LIBRARIES
%%local
sqlResults
%matplotlib inline
import numpy as np

# PLOT THE RESULTS
ax = sqlResults.plot(kind='scatter', figsize = (6,6), x='tip_amount', y='prediction', color='blue', alpha =
0.25, label='Actual vs. predicted');
fit = np.polyfit(sqlResults['tip_amount'], sqlResults['prediction'], deg=1)
ax.set_title('Actual vs. Predicted Tip Amounts ($)')
ax.set_xlabel("Actual")
ax.set_ylabel("Predicted")
#ax.plot(sqlResults['tip_amount'], fit[0] * sqlResults['prediction'] + fit[1], color='magenta')
plt.axis([-1, 15, -1, 8])
plt.show(ax)
```

Output:

Create a GBT regression model

Create a GBT regression model by using the Spark ML `GBTRegressor()` function, and then evaluate the model on test data.

Gradient-boosted trees (GBTS) are ensembles of decision trees. GBTS trains decision trees iteratively to minimize a loss function. You can use GBTS for regression and classification. They can handle categorical features, do not require feature scaling, and can capture nonlinearities and feature interactions. You also can use them in a multiclass-classification setting.

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# TRAIN A GBT REGRESSION MODEL
val gbt = new GBTRegressor().setLabelCol("label").setFeaturesCol("featuresCat").setMaxIter(10)
val gbtModel = gbt.fit(indexedTRAINwithCatFeat)

# MAKE PREDICTIONS
val predictions = gbtModel.transform(indexedTESTwithCatFeat)

# COMPUTE TEST SET R2
val evaluator = new
RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("r2")
val Test_R2 = evaluator.evaluate(predictions)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# PRINT THE RESULTS
println("Test R-sqr is: " + Test_R2);
```

Output:

Test R-sqr is: 0.7655383534596654

Advanced modeling utilities for optimization

In this section, you use machine learning utilities that developers frequently use for model optimization. Specifically, you can optimize machine learning models three different ways by using parameter sweeping and cross-validation:

- Split the data into train and validation sets, optimize the model by using hyper-parameter sweeping on a training set, and evaluate on a validation set (linear regression)
- Optimize the model by using cross-validation and hyper-parameter sweeping by using Spark ML's CrossValidator function (binary classification)
- Optimize the model by using custom cross-validation and parameter-sweeping code to use any machine learning function and parameter set (linear regression)

Cross-validation is a technique that assesses how well a model trained on a known set of data will generalize to predict the features of data sets on which it has not been trained. The general idea behind this technique is that a model is trained on a data set of known data, and then the accuracy of its predictions is tested against an independent data set. A common implementation is to divide a data set into *k*-folds, and then train the model in a round-robin fashion on all but one of the folds.

Hyper-parameter optimization is the problem of choosing a set of hyper-parameters for a learning algorithm, usually with the goal of optimizing a measure of the algorithm's performance on an independent data set. A hyper-parameter is a value that you must specify outside the model training procedure. Assumptions about hyper-parameter values can affect the flexibility and accuracy of the model. Decision trees have hyper-parameters, for example, such as the desired depth and number of leaves in the tree. You must set a misclassification penalty term for a support vector machine (SVM).

A common way to perform hyper-parameter optimization is to use a grid search, also called a **parameter sweep**. In a grid search, an exhaustive search is performed through the values of a specified subset of the hyper-parameter space for a learning algorithm. Cross-validation can supply a performance metric to sort out the optimal results produced by the grid search algorithm. If you use cross-validation hyper-parameter sweeping, you can help limit problems like overfitting a model to training data. This way, the model retains the

capacity to apply to the general set of data from which the training data was extracted.

Optimize a linear regression model with hyper-parameter sweeping

Next, split data into train and validation sets, use hyper-parameter sweeping on a training set to optimize the model, and evaluate on a validation set (linear regression).

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# RENAME `tip_amount` AS A LABEL
val OneHotTRAINLabeled =
OneHotTRAIN.select("tip_amount","features").withColumnRenamed(existingName="tip_amount",newName="label")
val OneHotTESTLabeled =
OneHotTEST.select("tip_amount","features").withColumnRenamed(existingName="tip_amount",newName="label")
OneHotTRAINLabeled.cache()
OneHotTESTLabeled.cache()

# DEFINE THE ESTIMATOR FUNCTION: `THE LinearRegression()` FUNCTION
val lr = new LinearRegression().setLabelCol("label").setFeaturesCol("features").setMaxIter(10)

# DEFINE THE PARAMETER GRID
val paramGrid = new ParamGridBuilder().addGrid(lr.regParam, Array(0.1, 0.01,
0.001)).addGrid(lr.fitIntercept).addGrid(lr.elasticNetParam, Array(0.1, 0.5, 0.9)).build()

# DEFINE THE PIPELINE WITH A TRAIN/TEST VALIDATION SPLIT (75% IN THE TRAINING SET), AND THEN THE SPECIFY
ESTIMATOR, EVALUATOR, AND PARAMETER GRID
val trainPct = 0.75
val trainValidationSplit = new TrainValidationSplit().setEstimator(lr).setEvaluator(new
RegressionEvaluator).setEstimatorParamMaps(paramGrid).setTrainRatio(trainPct)

# RUN THE TRAIN VALIDATION SPLIT AND CHOOSE THE BEST SET OF PARAMETERS
val model = trainValidationSplit.fit(OneHotTRAINLabeled)

# MAKE PREDICTIONS ON THE TEST DATA BY USING THE MODEL WITH THE COMBINATION OF PARAMETERS THAT PERFORMS THE
BEST
val testResults = model.transform(OneHotTESTLabeled).select("label", "prediction")

# COMPUTE TEST SET R2
val evaluator = new
RegressionEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("r2")
val Test_R2 = evaluator.evaluate(testResults)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

println("Test R-sqr is: " + Test_R2);
```

Output:

Test R-sqr is: 0.6226484708501209

Optimize the binary classification model by using cross-validation and hyper-parameter sweeping

This section shows you how to optimize a binary classification model by using cross-validation and hyper-parameter sweeping. This uses the Spark ML `CrossValidator` function.

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# CREATE DATA FRAMES WITH PROPERLY LABELED COLUMNS TO USE WITH THE TRAIN AND TEST SPLIT
val indexedTRAINwithCatFeatBinTargetRF =
indexedTRAINwithCatFeatBinTarget.select("labelBin","featuresCat").withColumnRenamed(existingName="labelBin",
newName="label").withColumnRenamed(existingName="featuresCat",newName="features")
val indexedTESTwithCatFeatBinTargetRF =
indexedTESTwithCatFeatBinTarget.select("labelBin","featuresCat").withColumnRenamed(existingName="labelBin",n
ewName="label").withColumnRenamed(existingName="featuresCat",newName="features")
indexedTRAINwithCatFeatBinTargetRF.cache()
indexedTESTwithCatFeatBinTargetRF.cache()

# DEFINE THE ESTIMATOR FUNCTION
val rf = new
RandomForestClassifier().setLabelCol("label").setFeaturesCol("features").setImpurity("gini").setSeed(1234).s
etFeatureSubsetStrategy("auto").setMaxBins(32)

# DEFINE THE PARAMETER GRID
val paramGrid = new ParamGridBuilder().addGrid(rf.maxDepth, Array(4,8)).addGrid(rf.numTrees,
Array(5,10)).addGrid(rf.minInstancesPerNode, Array(100,300)).build()

# SPECIFY THE NUMBER OF FOLDS
val numFolds = 3

# DEFINE THE TRAIN/TEST VALIDATION SPLIT (75% IN THE TRAINING SET)
val CrossValidator = new CrossValidator().setEstimator(rf).setEvaluator(new
BinaryClassificationEvaluator).setEstimatorParamMaps(paramGrid).setNumFolds(numFolds)

# RUN THE TRAIN VALIDATION SPLIT AND CHOOSE THE BEST SET OF PARAMETERS
val model = CrossValidator.fit(indexedTRAINwithCatFeatBinTargetRF)

# MAKE PREDICTIONS ON THE TEST DATA BY USING THE MODEL WITH THE COMBINATION OF PARAMETERS THAT PERFORMS THE
BEST
val testResults = model.transform(indexedTESTwithCatFeatBinTargetRF).select("label", "prediction")

# COMPUTE THE TEST F1 SCORE
val evaluator = new
MulticlassClassificationEvaluator().setLabelCol("label").setPredictionCol("prediction").setMetricName("f1")
val Test_f1Score = evaluator.evaluate(testResults)

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");
```

Output:

Time to run the cell: 33 seconds.

Optimize the linear regression model by using custom cross-validation and parameter-sweeping code

Next, optimize the model by using custom code, and identify the best model parameters by using the criterion of highest accuracy. Then, create the final model, evaluate the model on test data, and save the model in Blob storage. Finally, load the model, score test data, and evaluate accuracy.

```
# RECORD THE START TIME
val starttime = Calendar.getInstance().getTime()

# DEFINE THE PARAMETER GRID AND THE NUMBER OF FOLDS
val paramGrid = new ParamGridBuilder().addGrid(rf.maxDepth, Array(5,10)).addGrid(rf.numTrees,
Array(10,25,50)).build()

val nFolds = 3
val numModels = paramGrid.size
val numParamsinGrid = 2
```

```
val numParamsinGrid = 2

# SPECIFY THE NUMBER OF CATEGORIES FOR CATEGORICAL VARIABLES
val categoricalFeaturesInfo = Map[Int, Int]((0,2),(1,2),(2,6),(3,4))

var maxDepth = -1
var numTrees = -1
var param = ""
var paramval = -1
var validateLB = -1.0
var validateUB = -1.0
val h = 1.0 / nFolds;
val RMSE  = Array.fill(numModels)(0.0)

# CREATE K-FOLDS
val splits = MLUtils.kFold(indexedTRAINbinary, numFolds = nFolds, seed=1234)

# LOOP THROUGH K-FOLDS AND THE PARAMETER GRID TO GET AND IDENTIFY THE BEST PARAMETER SET BY LEVEL OF
ACCURACY
for (i <- 0 to (nFolds-1)) {
    validateLB = i * h
    validateUB = (i + 1) * h
    val validationCV = trainData.filter($"rand" >= validateLB  && $"rand" < validateUB)
    val trainCV = trainData.filter($"rand" < validateLB  || $"rand" >= validateUB)
    val validationLabPt = validationCV.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)));
    val trainCVLabPt = trainCV.rdd.map(r => LabeledPoint(r.getDouble(targetIndRegression(0).toInt),
Vectors.dense(featuresIndIndex.map(r.getDouble(_)).toArray)));
    validationLabPt.cache()
    trainCVLabPt.cache()

    for (nParamSets <- 0 to (numModels-1)) {
        for (nParams <- 0 to (numParamsinGrid-1)) {
            param = paramGrid(nParamSets).toSeq(nParams).param.toString.split("__")(1)
            paramval = paramGrid(nParamSets).toSeq(nParams).value.toString.toInt
            if (param == "maxDepth") {maxDepth = paramval}
            if (param == "numTrees") {numTrees = paramval}
        }
        val rfModel = RandomForest.trainRegressor(trainCVLabPt,
categoricalFeaturesInfo=categoricalFeaturesInfo,
                                                  numTrees=numTrees, maxDepth=maxDepth,
                                                  featureSubsetStrategy="auto",impurity="variance",
maxBins=32)
        val labelAndPreds = validationLabPt.map { point =>
                                                 val prediction = rfModel.predict(point.features)
                                                  ( prediction, point.label )
                                                }
        val validMetrics = new RegressionMetrics(labelAndPreds)
        val rmse = validMetrics.rootMeanSquaredError
        RMSE(nParamSets) += rmse
    }
    validationLabPt.unpersist();
    trainCVLabPt.unpersist();
}
val minRMSEindex = RMSE.indexOf(RMSE.min)

# GET THE BEST PARAMETERS FROM A CROSS-VALIDATION AND PARAMETER SWEEP
var best_maxDepth = -1
var best_numTrees = -1
for (nParams <- 0 to (numParamsinGrid-1)) {
    param = paramGrid(minRMSEindex).toSeq(nParams).param.toString.split("__")(1)
    paramval = paramGrid(minRMSEindex).toSeq(nParams).value.toString.toInt
    if (param == "maxDepth") {best_maxDepth = paramval}
    if (param == "numTrees") {best_numTrees = paramval}
}

# CREATE THE BEST MODEL WITH THE BEST PARAMETERS AND A FULL TRAINING DATA SET
val best_rfModel = RandomForest.trainRegressor(indexedTRAINreg,
categoricalFeaturesInfo=categoricalFeaturesInfo,
                                               numTrees=best_numTrees, maxDepth=best_maxDepth
```

```
                                                numTrees=best_numTrees, maxDepth=best_maxDepth,
                                                featureSubsetStrategy="auto",impurity="variance",
maxBins=32)

# SAVE THE BEST RANDOM FOREST MODEL IN BLOB STORAGE
val datestamp = Calendar.getInstance().getTime().toString.replaceAll(" ", ".").replaceAll(":", "_");
val modelName = "BestCV_RF_Regression__"
val filename = modelDir.concat(modelName).concat(datestamp)
best_rfModel.save(sc, filename);

# PREDICT ON THE TRAINING SET WITH THE BEST MODEL AND THEN EVALUATE
val labelAndPreds = indexedTESTreg.map { point =>
                                         val prediction = best_rfModel.predict(point.features)
                                         ( prediction, point.label )
                                        }

val test_rmse = new RegressionMetrics(labelAndPreds).rootMeanSquaredError
val test_rsqr = new RegressionMetrics(labelAndPreds).r2

# GET THE TIME TO RUN THE CELL
val endtime = Calendar.getInstance().getTime()
val elapsedtime =  ((endtime.getTime() - starttime.getTime())/1000).toString;
println("Time taken to run the above cell: " + elapsedtime + " seconds.");

# LOAD THE MODEL
val savedRFModel = RandomForestModel.load(sc, filename)

val labelAndPreds = indexedTESTreg.map { point =>
                                         val prediction = savedRFModel.predict(point.features)
                                         ( prediction, point.label )
                                        }
# TEST THE MODEL
val test_rmse = new RegressionMetrics(labelAndPreds).rootMeanSquaredError
val test_rsqr = new RegressionMetrics(labelAndPreds).r2
```

Output:

Time to run the cell: 61 seconds.

Consume Spark-built machine learning models automatically with Scala

For an overview of topics that walk you through the tasks that comprise the Data Science process in Azure, see Team Data Science Process.

Team Data Science Process walkthroughs describes other end-to-end walkthroughs that demonstrate the steps in the Team Data Science Process for specific scenarios. The walkthroughs also illustrate how to combine cloud and on-premises tools and services into a workflow or pipeline to create an intelligent application.

Score Spark-built machine learning models shows you how to use Scala code to automatically load and score new data sets with machine learning models built in Spark and saved in Azure Blob storage. You can follow the instructions provided there, and simply replace the Python code with Scala code in this article for automated consumption.

Feature engineering in machine learning

11/19/2021 • 6 minutes to read • Edit Online

In this article, you learn about feature engineering and its role in enhancing data in machine learning. Learn from illustrative examples drawn from Azure Machine Learning Studio (classic) experiments.

- **Feature engineering**: The process of creating new features from raw data to increase the predictive power of the learning algorithm. Engineered features should capture additional information that is not easily apparent in the original feature set.
- **Feature selection**: The process of selecting the key subset of features to reduce the dimensionality of the training problem.

Normally **feature engineering** is applied first to generate additional features, and then **feature selection** is done to eliminate irrelevant, redundant, or highly correlated features.

Feature engineering and selection are part of the modeling stage of the Team Data Science Process (TDSP). To learn more about the TDSP and the data science lifecycle, see What is the TDSP?

What is feature engineering?

Training data consists of a matrix composed of rows and columns. Each row in the matrix is an observation or record. The columns of each row are the features that describe each record. The features specified in the experimental design should characterize the patterns in the data.

Although many of the raw data fields can be used directly to train a model, it's often necessary to create additional (engineered) features for an enhanced training dataset.

Engineered features that enhance training provide information that better differentiates the patterns in the data. But this process is something of an art. Sound and productive decisions often require domain expertise.

Example 1: Add temporal features for a regression model

Let's use the experiment Demand forecasting of bikes rentals in Azure Machine Learning Studio (classic) to demonstrate how to engineer features for a regression task. The objective of this experiment is to predict the demand for bike rentals within a specific month/day/hour.

Bike rental dataset

The Bike Rental UCI dataset is based on real data from a bike share company based in the United States. It represents the number of bike rentals within a specific hour of a day for the years 2011 and 2012. It contains 17,379 rows and 17 columns.

The raw feature set contains weather conditions (temperature/humidity/wind speed) and the type of the day (holiday/weekday). The field to predict is the count, which represents the bike rentals within a specific hour. Count ranges from 1 to 977.

Create a feature engineering experiment

With the goal of constructing effective features in the training data, four regression models are built using the same algorithm but with four different training datasets. The four datasets represent the same raw input data, but with an increasing number of features set. These features are grouped into four categories:

1. A = weather + holiday + weekday + weekend features for the predicted day
2. B = number of bikes that were rented in each of the previous 12 hours

3. C = number of bikes that were rented in each of the previous 12 days at the same hour
4. D = number of bikes that were rented in each of the previous 12 weeks at the same hour and the same day

Besides feature set A, which already exists in the original raw data, the other three sets of features are created through the feature engineering process. Feature set B captures recent demand for the bikes. Feature set C captures the demand for bikes at a particular hour. Feature set D captures demand for bikes at particular hour and particular day of the week. The four training datasets each includes feature set A, A+B, A+B+C, and A+B+C+D, respectively.

Feature engineering using Studio (classic)

In the Studio (classic) experiment, these four training datasets are formed via four branches from the pre-processed input dataset. Except for the leftmost branch, each of these branches contains an Execute R Script module, in which the derived features (feature set B, C, and D) are constructed and appended to the imported dataset.

The following figure demonstrates the R script used to create feature set B in the second left branch.

Results

A comparison of the performance results of the four models is summarized in the following table:

Features	Mean Absolute Error	Root Mean Square Error
A	89.7	124.9
A + B	51.7	88.3
A + B + C	47.6	81.1
A + B + C + D	48.3	82.1

The best results are shown by features A+B+C. The error rate decreases when additional feature set are included in the training data. It verifies the presumption that the feature set B, C provide additional relevant information for the regression task. But adding the D feature does not seem to provide any additional reduction in the error rate.

Example 2: Create features for text mining

Feature engineering is widely applied in tasks related to text mining such as document classification and sentiment analysis. Since individual pieces of raw text usually serve as the input data, the feature engineering process is needed to create the features involving word/phrase frequencies.

Feature hashing

To achieve this task, a technique called feature hashing is applied to efficiently turn arbitrary text features into indices. Instead of associating each text feature (words/phrases) to a particular index, this method applies a hash function to the features and using their hash values as indices directly.

In Studio (classic), there is a Feature Hashing module that creates word/phrase features conveniently. Following figure shows an example of using this module. The input dataset contains two columns: the book rating ranging from 1 to 5, and the actual review content. The goal of this module is to retrieve a bunch of new features that

show the occurrence frequency of the corresponding word(s)/phrase(s) within the particular book review. To use this module, complete the following steps:

- First, select the column that contains the input text ("Col2" in this example).
- Second, set the "Hashing bitsize" to 8, which means 2^8=256 features will be created. The word/phase in all the text will be hashed to 256 indices. The parameter "Hashing bitsize" ranges from 1 to 31. The word(s)/phrase(s) are less likely to be hashed into the same index if setting it to be a larger number.
- Third, set the parameter "N-grams" to 2. This value gets the occurrence frequency of unigrams (a feature for every single word) and bigrams (a feature for every pair of adjacent words) from the input text. The parameter "N-grams" ranges from 0 to 10, which indicates the maximum number of sequential words to be included in a feature.

The following figure shows what these new feature look like.

Conclusion

Engineered and selected features increase the efficiency of the training process, which attempts to extract the key information contained in the data. They also improve the power of these models to classify the input data accurately and to predict outcomes of interest more robustly.

Feature engineering and selection can also combine to make the learning more computationally tractable. It does so by enhancing and then reducing the number of features needed to calibrate or train a model. Mathematically, the selected features are a minimal set of independent variables that explain the patterns in the data and predict outcomes successfully.

It's not always necessarily to perform feature engineering or feature selection. It depends on the data, the algorithm selected, and the objective of the experiment.

Next steps

To create features for data in specific environments, see the following articles:

- Create features for data in SQL Server
- Create features for data in a Hadoop cluster using Hive queries

Create features for data in SQL Server using SQL and Python

11/19/2021 • 5 minutes to read • Edit Online

This document shows how to generate features for data stored in a SQL Server VM on Azure that help algorithms learn more efficiently from the data. You can use SQL or a programming language like Python to accomplish this task. Both approaches are demonstrated here.

This task is a step in the Team Data Science Process (TDSP).

> **NOTE**
> For a practical example, you can consult the NYC Taxi dataset and refer to the IPNB titled NYC Data wrangling using IPython Notebook and SQL Server for an end-to-end walk-through.

Prerequisites

This article assumes that you have:

- Created an Azure storage account. If you need instructions, see Create an Azure Storage account
- Stored your data in SQL Server. If you have not, see Move data to an Azure SQL Database for Azure Machine Learning for instructions on how to move the data there.

Feature generation with SQL

In this section, we describe ways of generating features using SQL:

- Count based Feature Generation
- Binning Feature Generation
- Rolling out the features from a single column

> **NOTE**
> Once you generate additional features, you can either add them as columns to the existing table or create a new table with the additional features and primary key, that can be joined with the original table.

Count based feature generation

This document demonstrates two ways of generating count features. The first method uses conditional sum and the second method uses the 'where` clause. These new features can then be joined with the original table (using primary key columns) to have count features alongside the original data.

```
select <column_name1>,<column_name2>,<column_name3>, COUNT(*) as Count_Features from <tablename> group by
<column_name1>,<column_name2>,<column_name3>

select <column_name1>,<column_name2> , sum(1) as Count_Features from <tablename>
where <column_name3> = '<some_value>' group by <column_name1>,<column_name2>
```

Binning Feature Generation

The following example shows how to generate binned features by binning (using five bins) a numerical column

that can be used as a feature instead:

```
SELECT <column_name>, NTILE(5) OVER (ORDER BY <column_name>) AS BinNumber from <tablename>
```

Rolling out the features from a single column

In this section, we demonstrate how to roll out a single column in a table to generate additional features. The example assumes that there is a latitude or longitude column in the table from which you are trying to generate features.

Here is a brief primer on latitude/longitude location data (resourced from stackoverflow `https://gis.stackexchange.com/questions/8650/how-to-measure-the-accuracy-of-latitude-and-longitude`). Here are some useful things to understand about location data before creating features from the field:

- The sign indicates whether we are north or south, east or west on the globe.
- A nonzero hundreds digit indicates longitude, not latitude is being used.
- The tens digit gives a position to about 1,000 kilometers. It gives useful information about what continent or ocean we are on.
- The units digit (one decimal degree) gives a position up to 111 kilometers (60 nautical miles, about 69 miles). It indicates, roughly, what large state or country/region we are in.
- The first decimal place is worth up to 11.1 km: it can distinguish the position of one large city from a neighboring large city.
- The second decimal place is worth up to 1.1 km: it can separate one village from the next.
- The third decimal place is worth up to 110 m: it can identify a large agricultural field or institutional campus.
- The fourth decimal place is worth up to 11 m: it can identify a parcel of land. It is comparable to the typical accuracy of an uncorrected GPS unit with no interference.
- The fifth decimal place is worth up to 1.1 m: it distinguishes trees from each other. Accuracy to this level with commercial GPS units can only be achieved with differential correction.
- The sixth decimal place is worth up to 0.11 m: you can use this level for laying out structures in detail, for designing landscapes, building roads. It should be more than good enough for tracking movements of glaciers and rivers. This goal can be achieved by taking painstaking measures with GPS, such as differentially corrected GPS.

The location information can be featurized by separating out region, location, and city information. Once can also call a REST endpoint, such as Bing Maps API (see `https://msdn.microsoft.com/library/ff701710.aspx` to get the region/district information).

```
select
    <location_columnname>
    ,round(<location_columnname>,0) as l1
    ,l2=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 1 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),1,1) else '0' end
    ,l3=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 2 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),2,1) else '0' end
    ,l4=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 3 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),3,1) else '0' end
    ,l5=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 4 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),4,1) else '0' end
    ,l6=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 5 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),5,1) else '0' end
    ,l7=case when LEN (PARSENAME(round(ABS(<location_columnname>) - FLOOR(ABS(<location_columnname>)),6),1))
>= 6 then substring(PARSENAME(round(ABS(<location_columnname>) -
FLOOR(ABS(<location_columnname>)),6),1),6,1) else '0' end
from <tablename>
```

These location-based features can be further used to generate additional count features as described earlier.

> **TIP**
> You can programmatically insert the records using your language of choice. You may need to insert the data in chunks to improve write efficiency. Here is an example of how to do this using pyodbc. Another alternative is to insert data in the database using BCP utility

Connecting to Azure Machine Learning

The newly generated feature can be added as a column to an existing table or stored in a new table and joined with the original table for machine learning. Features can be generated or accessed if already created, using the Import Data module in Azure ML as shown below:

Using a programming language like Python

Using Python to generate features when the data is in SQL Server is similar to processing data in Azure blob using Python. For comparison, see Process Azure Blob data in your data science environment. Load the data

from the database into a pandas data frame to process it further. The process of connecting to the database and loading the data into the data frame is documented in this section.

The following connection string format can be used to connect to a SQL Server database from Python using pyodbc (replace servername, dbname, username, and password with your specific values):

```
#Set up the SQL Azure connection
import pyodbc
conn = pyodbc.connect('DRIVER={SQL Server};SERVER=<servername>;DATABASE=<dbname>;UID=<username>;PWD=
<password>')
```

The Pandas library in Python provides a rich set of data structures and data analysis tools for data manipulation for Python programming. The following code reads the results returned from a SQL Server database into a Pandas data frame:

```
# Query database and load the returned results in pandas data frame
data_frame = pd.read_sql('''select <columnname1>, <columnname2>... from <tablename>''', conn)
```

Now you can work with the Pandas data frame as covered in topics Create features for Azure blob storage data using Panda.

Create features for data in a Hadoop cluster using Hive queries

11/19/2021 • 7 minutes to read • Edit Online

This document shows how to create features for data stored in an Azure HDInsight Hadoop cluster using Hive queries. These Hive queries use embedded Hive User-Defined Functions (UDFs), the scripts for which are provided.

The operations needed to create features can be memory intensive. The performance of Hive queries becomes more critical in such cases and can be improved by tuning certain parameters. The tuning of these parameters is discussed in the final section.

Examples of the queries that are presented are specific to the NYC Taxi Trip Data scenarios are also provided in GitHub repository. These queries already have data schema specified and are ready to be submitted to run. In the final section, parameters that users can tune so that the performance of Hive queries can be improved are also discussed.

This task is a step in the Team Data Science Process (TDSP).

Prerequisites

This article assumes that you have:

- Created an Azure storage account. If you need instructions, see Create an Azure Storage account
- Provisioned a customized Hadoop cluster with the HDInsight service. If you need instructions, see Customize Azure HDInsight Hadoop Clusters for Advanced Analytics.
- The data has been uploaded to Hive tables in Azure HDInsight Hadoop clusters. If it has not, follow Create and load data to Hive tables to upload data to Hive tables first.
- Enabled remote access to the cluster. If you need instructions, see Access the Head Node of Hadoop Cluster.

Feature generation

In this section, several examples of the ways in which features can be generating using Hive queries are described. Once you have generated additional features, you can either add them as columns to the existing table or create a new table with the additional features and primary key, which can then be joined with the original table. Here are the examples presented:

1. Frequency-based Feature Generation
2. Risks of Categorical Variables in Binary Classification
3. Extract features from Datetime Field
4. Extract features from Text Field
5. Calculate distance between GPS coordinates

Frequency-based feature generation

It is often useful to calculate the frequencies of the levels of a categorical variable, or the frequencies of certain combinations of levels from multiple categorical variables. Users can use the following script to calculate these frequencies:

```
select
    a.<column_name1>, a.<column_name2>, a.sub_count/sum(a.sub_count) over () as frequency
from
(
    select
        <column_name1>,<column_name2>, count(*) as sub_count
    from <databasename>.<tablename> group by <column_name1>, <column_name2>
)a
order by frequency desc;
```

Risks of categorical variables in binary classification

In binary classification, non-numeric categorical variables must be converted into numeric features when the models being used only take numeric features. This conversion is done by replacing each non-numeric level with a numeric risk. This section shows some generic Hive queries that calculate the risk values (log odds) of a categorical variable.

```
set smooth_param1=1;
set smooth_param2=20;
select
    <column_name1>,<column_name2>,
    ln((sum_target+${hiveconf:smooth_param1})/(record_count-sum_target+${hiveconf:smooth_param2}-
${hiveconf:smooth_param1})) as risk
from
    (
    select
        <column_nam1>, <column_name2>, sum(binary_target) as sum_target, sum(1) as record_count
    from
        (
        select
            <column_name1>, <column_name2>, if(target_column>0,1,0) as binary_target
        from <databasename>.<tablename>
        )a
    group by <column_name1>, <column_name2>
    )b
```

In this example, variables `smooth_param1` and `smooth_param2` are set to smooth the risk values calculated from the data. Risks have a range between -Inf and Inf. A risk > 0 indicates that the probability that the target is equal to 1 is greater than 0.5.

After the risk table is calculated, users can assign risk values to a table by joining it with the risk table. The Hive joining query was provided in previous section.

Extract features from datetime fields

Hive comes with a set of UDFs for processing datetime fields. In Hive, the default datetime format is 'yyyy-MM-dd 00:00:00' ('1970-01-01 12:21:32' for example). This section shows examples that extract the day of a month, the month from a datetime field, and other examples that convert a datetime string in a format other than the default format to a datetime string in default format.

```
select day(<datetime field>), month(<datetime field>)
from <databasename>.<tablename>;
```

This Hive query assumes that the *<datetime field>* is in the default datetime format.

If a datetime field is not in the default format, you need to convert the datetime field into Unix time stamp first, and then convert the Unix time stamp to a datetime string that is in the default format. When the datetime is in default format, users can apply the embedded datetime UDFs to extract features.

```
select from_unixtime(unix_timestamp(<datetime field>,'<pattern of the datetime field>'))
from <databasename>.<tablename>;
```

In this query, if the *<datetime field>* has the pattern like *03/26/2015 12:04:39*, the *<pattern of the datetime field>* '' should be `'MM/dd/yyyy HH:mm:ss'`. To test it, users can run

```
select from_unixtime(unix_timestamp('05/15/2015 09:32:10','MM/dd/yyyy HH:mm:ss'))
from hivesampletable limit 1;
```

The *hivesampletable* in this query comes preinstalled on all Azure HDInsight Hadoop clusters by default when the clusters are provisioned.

Extract features from text fields

When the Hive table has a text field that contains a string of words that are delimited by spaces, the following query extracts the length of the string, and the number of words in the string.

```
select length(<text field>) as str_len, size(split(<text field>,' ')) as word_num
from <databasename>.<tablename>;
```

Calculate distances between sets of GPS coordinates

The query given in this section can be directly applied to the NYC Taxi Trip Data. The purpose of this query is to show how to apply an embedded mathematical function in Hive to generate features.

The fields that are used in this query are the GPS coordinates of pickup and dropoff locations, named *pickup_longitude*, *pickup_latitude*, *dropoff_longitude*, and *dropoff_latitude*. The queries that calculate the direct distance between the pickup and dropoff coordinates are:

```
set R=3959;
set pi=radians(180);
select pickup_longitude, pickup_latitude, dropoff_longitude, dropoff_latitude,
    ${hiveconf:R}*2*2*atan((1-sqrt(1-pow(sin((dropoff_latitude-pickup_latitude)
    *${hiveconf:pi}/180/2),2)-cos(pickup_latitude*${hiveconf:pi}/180)
    *cos(dropoff_latitude*${hiveconf:pi}/180)*pow(sin((dropoff_longitude-
pickup_longitude)*${hiveconf:pi}/180/2),2)))
    /sqrt(pow(sin((dropoff_latitude-pickup_latitude)*${hiveconf:pi}/180/2),2)
    +cos(pickup_latitude*${hiveconf:pi}/180)*cos(dropoff_latitude*${hiveconf:pi}/180)*
    pow(sin((dropoff_longitude-pickup_longitude)*${hiveconf:pi}/180/2),2))) as direct_distance
from nyctaxi.trip
where pickup_longitude between -90 and 0
and pickup_latitude between 30 and 90
and dropoff_longitude between -90 and 0
and dropoff_latitude between 30 and 90
limit 10;
```

The mathematical equations that calculate the distance between two GPS coordinates can be found on the Movable Type Scripts site, authored by Peter Lapisu. In this Javascript, the function `toRad()` is just *lat_or_lon*pi/180, which converts degrees to radians. Here, *lat_or_lon* is the latitude or longitude. Since Hive does not provide the function `atan2`, but provides the function `atan`, the `atan2` function is implemented by `atan` function in the above Hive query using the definition provided in Wikipedia.

$$\operatorname{atan2}(y, x) = 2\arctan\frac{\sqrt{x^2+y^2}-x}{y}.$$

A full list of Hive embedded UDFs can be found in the **Built-in Functions** section on the Apache Hive wiki).

Advanced topics: Tune Hive parameters to improve query speed

The default parameter settings of Hive cluster might not be suitable for the Hive queries and the data that the queries are processing. This section discusses some parameters that users can tune to improve the performance of Hive queries. Users need to add the parameter tuning queries before the queries of processing data.

1. **Java heap space**: For queries involving joining large datasets, or processing long records, **running out of heap space** is one of the common errors. This error can be avoided by setting parameters *mapreduce.map.java.opts* and *mapreduce.task.io.sort.mb* to desired values. Here is an example:

   ```
   set mapreduce.map.java.opts=-Xmx4096m;
   set mapreduce.task.io.sort.mb=-Xmx1024m;
   ```

   This parameter allocates 4-GB memory to Java heap space and also makes sorting more efficient by allocating more memory for it. It is a good idea to play with these allocations if there are any job failure errors related to heap space.

2. **DFS block size**: This parameter sets the smallest unit of data that the file system stores. As an example, if the DFS block size is 128 MB, then any data of size less than and up to 128 MB is stored in a single block. Data that is larger than 128 MB is allotted extra blocks.

3. Choosing a small block size causes large overheads in Hadoop since the name node has to process many more requests to find the relevant block pertaining to the file. A recommended setting when dealing with gigabytes (or larger) data is:

   ```
   set dfs.block.size=128m;
   ```

4. **Optimizing join operation in Hive**: While join operations in the map/reduce framework typically take place in the reduce phase, sometimes, enormous gains can be achieved by scheduling joins in the map phase (also called "mapjoins"). Set this option:

   ```
   set hive.auto.convert.join=true;
   ```

5. **Specifying the number of mappers to Hive**: While Hadoop allows the user to set the number of reducers, the number of mappers is typically not be set by the user. A trick that allows some degree of control on this number is to choose the Hadoop variables *mapred.min.split.size* and *mapred.max.split.size* as the size of each map task is determined by:

   ```
   num_maps = max(mapred.min.split.size, min(mapred.max.split.size, dfs.block.size))
   ```

   Typically, the default value of:

   - *mapred.min.split.size* is 0, that of
   - *mapred.max.split.size* is **Long.MAX** and that of
   - *dfs.block.size* is 64 MB.

   As we can see, given the data size, tuning these parameters by "setting" them allows us to tune the number of mappers used.

6. Here are a few other more **advanced options** for optimizing Hive performance. These options allow you to set the memory allocated to map and reduce tasks, and can be useful in tweaking performance. Keep in mind that the *mapreduce.reduce.memory.mb* cannot be greater than the physical memory size of each worker node in the Hadoop cluster.

```
set mapreduce.map.memory.mb = 2048;
set mapreduce.reduce.memory.mb=6144;
set mapreduce.reduce.java.opts=-Xmx8192m;
set mapred.reduce.tasks=128;
set mapred.tasktracker.reduce.tasks.maximum=128;
```

Feature selection in the Team Data Science Process (TDSP)

11/19/2021 • 3 minutes to read • Edit Online

This article explains the purposes of feature selection and provides examples of its role in the data enhancement process of machine learning. These examples are drawn from Azure Machine Learning Studio.

The engineering and selection of features is one part of the Team Data Science Process (TDSP) outlined in the article What is the Team Data Science Process?. Feature engineering and selection are parts of the **Develop features** step of the TDSP.

- **feature engineering**: This process attempts to create additional relevant features from the existing raw features in the data, and to increase predictive power to the learning algorithm.
- **feature selection**: This process selects the key subset of original data features in an attempt to reduce the dimensionality of the training problem.

Normally **feature engineering** is applied first to generate additional features, and then the **feature selection** step is performed to eliminate irrelevant, redundant, or highly correlated features.

Filter features from your data - feature selection

Feature selection may be used for classification or regression tasks. The goal is to select a subset of the features from the original dataset that reduce its dimensions by using a minimal set of features to represent the maximum amount of variance in the data. This subset of features is used to train the model. Feature selection serves two main purposes.

- First, feature selection often increases classification accuracy by eliminating irrelevant, redundant, or highly correlated features.
- Second, it decreases the number of features, which makes the model training process more efficient. Efficiency is important for learners that are expensive to train such as support vector machines.

Although feature selection does seek to reduce the number of features in the dataset used to train the model, it is not referred to by the term "dimensionality reduction". Feature selection methods extract a subset of original features in the data without changing them. Dimensionality reduction methods employ engineered features that can transform the original features and thus modify them. Examples of dimensionality reduction methods include principal component analysis (PCA), canonical correlation analysis, and singular value decomposition (SVD).

Among others, one widely applied category of feature selection methods in a supervised context is called "filter-based feature selection". By evaluating the correlation between each feature and the target attribute, these methods apply a statistical measure to assign a score to each feature. The features are then ranked by the score, which may be used to help set the threshold for keeping or eliminating a specific feature. Examples of statistical measures used in these methods include Pearson correlation coefficient (PCC), mutual information (MI), and the chi-squared test.

In Azure Machine Learning Studio, there are modules provided for feature selection. As shown in the following figure, these modules include Filter-Based Feature Selection and Fisher Linear Discriminant Analysis.

Consider, for example, the use of the Filter-Based Feature Selection module. For convenience, continue using the text mining example. Assume that one wants to build a regression model after a set of 256 features are created through the Feature Hashing module, and that the response variable is the "Col1" that contains book review ratings ranging from 1 to 5. By setting "Feature scoring method" to be "Pearson Correlation", the "Target column" to be "Col1", and the "Number of desired features" to 50. Then the module Filter-Based Feature Selection produces a dataset containing 50 features together with the target attribute "Col1". The following figure shows the flow of this experiment and the input parameters:

The following figure shows the resulting datasets:

Each feature is scored based on the Pearson correlation coefficient between itself and the target attribute "Col1". The features with top scores are kept.

The corresponding scores of the selected features are shown in the following figure:

By applying this Filter-Based Feature Selection module, 50 out of 256 features are selected because they have the most correlated features with the target variable "Col1", based on the scoring method "Pearson Correlation".

Conclusion

Feature engineering and feature selection are two commonly engineering techniques to increase training efficiency. These techniques also improve the model's power to classify the input data accurately and to predict outcomes of interest more robustly. Feature engineering and selection can also combine to make the learning more computationally efficient by enhancing and then reducing the number of features needed to calibrate or train a model. Mathematically speaking, the features selected to train the model are a minimal set of independent variables that explain the maximum variance in the data to predict the outcome feature.

It is not always necessarily to perform feature engineering or feature selection. Whether it is needed or not depends on the data collected, the algorithm selected, and the objective of the experiment.

Deploy ML models to production to play an active role in making business decisions

11/19/2021 • 2 minutes to read • Edit Online

Production deployment enables a model to play an active role in a business. Predictions from a deployed model can be used for business decisions.

Production platforms

There are various approaches and platforms to put models into production. Here are a few options:

- Where to deploy models with Azure Machine Learning
- Deployment of a model in SQL-server
- Microsoft Machine Learning Server

> **NOTE**
> Prior to deployment, one has to insure the latency of model scoring is low enough to use in production.

> **NOTE**
> For deployment using Azure Machine Learning Studio, see Deploy an Azure Machine Learning web service.

A/B testing

When multiple models are in production, A/B testing may be used to compare model performance.

Next steps

Walkthroughs that demonstrate all the steps in the process for **specific scenarios** are also provided. They are listed and linked with thumbnail descriptions in the Example walkthroughs article. They illustrate how to combine cloud, on-premises tools, and services into a workflow or pipeline to create an intelligent application.

Machine Learning Anomaly Detection API

11/19/2021 • 10 minutes to read • Edit Online

> **NOTE**
>
> This item is under maintenance. We encourage you to use the Anomaly Detector API service powered by a gallery of Machine Learning algorithms under Azure Cognitive Services to detect anomalies from business, operational, and IoT metrics.

Overview

Anomaly Detection API is an example built with Azure Machine Learning that detects anomalies in time series data with numerical values that are uniformly spaced in time.

This API can detect the following types of anomalous patterns in time series data:

- **Positive and negative trends**: For example, when monitoring memory usage in computing an upward trend may be of interest as it may be indicative of a memory leak,
- **Changes in the dynamic range of values**: For example, when monitoring the exceptions thrown by a cloud service, any changes in the dynamic range of values could indicate instability in the health of the service, and
- **Spikes and Dips**: For example, when monitoring the number of login failures in a service or number of checkouts in an e-commerce site, spikes or dips could indicate abnormal behavior.

These machine learning detectors track such changes in values over time and report ongoing changes in their values as anomaly scores. They do not require adhoc threshold tuning and their scores can be used to control false positive rate. The anomaly detection API is useful in several scenarios like service monitoring by tracking KPIs over time, usage monitoring through metrics such as number of searches, numbers of clicks, performance monitoring through counters like memory, CPU, file reads, etc. over time.

The Anomaly Detection offering comes with useful tools to get you started.

- The web application helps you evaluate and visualize the results of anomaly detection APIs on your data.

> **NOTE**
>
> Try **IT Anomaly Insights solution** powered by this API

API Deployment

In order to use the API, you must deploy it to your Azure subscription where it will be hosted as an Azure Machine Learning web service. You can do this from the Azure AI Gallery. This will deploy two Azure Machine Learning Studio (classic) Web Services (and their related resources) to your Azure subscription - one for anomaly detection with seasonality detection, and one without seasonality detection. Once the deployment has completed, you will be able to manage your APIs from the Azure Machine Learning Studio (classic) web services page. From this page, you will be able to find your endpoint locations, API keys, as well as sample code for calling the API. More detailed instructions are available here.

Scaling the API

By default, your deployment will have a free Dev/Test billing plan that includes 1,000 transactions/month and 2 compute hours/month. You can upgrade to another plan as per your needs. Details on the pricing of different plans are available here under "Production Web API pricing".

Managing AML plans

You can manage your billing plan here. The plan name will be based on the resource group name you chose when deploying the API, plus a string that is unique to your subscription. Instructions on how to upgrade your plan are available here under the "Managing billing plans" section.

API definition

The web service provides a REST-based API over HTTPS that can be consumed in different ways including a web or mobile application, R, Python, Excel, etc. You send your time series data to this service via a REST API call, and it runs a combination of the three anomaly types described below.

Calling the API

In order to call the API, you will need to know the endpoint location and API key. These two requirements, along with sample code for calling the API, are available from the Azure Machine Learning Studio (classic) web services page. Navigate to the desired API, and then click the "Consume" tab to find them. You can call the API as a Swagger API (that is, with the URL parameter `format=swagger`) or as a non-Swagger API (that is, without the `format` URL parameter). The sample code uses the Swagger format. Below is an example request and response in non-Swagger format. These examples are to the seasonality endpoint. The non-seasonality endpoint is similar.

Sample request body

The request contains two objects: `Inputs` and `GlobalParameters`. In the example request below, some parameters are sent explicitly while others are not (scroll down for a full list of parameters for each endpoint). Parameters that are not sent explicitly in the request will use the default values given below.

```
{
    "Inputs": {
        "input1": {
            "ColumnNames": ["Time", "Data"],
            "Values": [
                ["5/30/2010 18:07:00", "1"],
                ["5/30/2010 18:08:00", "1.4"],
                ["5/30/2010 18:09:00", "1.1"]
            ]
        }
    },
    "GlobalParameters": {
        "tspikedetector.sensitivity": "3",
        "zspikedetector.sensitivity": "3",
        "bileveldetector.sensitivity": "3.25",
        "detectors.spikesdips": "Both"
    }
}
```

Sample response

In order to see the `ColumnNames` field, you must include `details=true` as a URL parameter in your request. See the tables below for the meaning behind each of these fields.

```
{
    "Results": {
        "output1": {
            "type": "table",
            "value": {
                "Values": [
                    ["5/30/2010 6:07:00 PM", "1", "1", "0", "0", "-0.687952590518378", "0", "-
0.687952590518378", "0", "-0.687952590518378", "0"],
                    ["5/30/2010 6:08:00 PM", "1.4", "1.4", "0", "0", "-1.07030497733224", "0", "-
0.884548154298423", "0", "-1.07030497733224", "0"],
                    ["5/30/2010 6:09:00 PM", "1.1", "1.1", "0", "0", "-1.30229513613974", "0", "-
1.173800281031", "0", "-1.30229513613974", "0"]
                ],
                "ColumnNames": ["Time", "OriginalData", "ProcessedData", "TSpike", "ZSpike",
"BiLevelChangeScore", "BiLevelChangeAlert", "PosTrendScore", "PosTrendAlert", "NegTrendScore",
"NegTrendAlert"],
                "ColumnTypes": ["DateTime", "Double", "Double", "Double", "Double", "Double", "Int32",
"Double", "Int32", "Double", "Int32"]
            }
        }
    }
}
```


Score API

The Score API is used for running anomaly detection on non-seasonal time series data. The API runs a number of anomaly detectors on the data and returns their anomaly scores. The figure below shows an example of anomalies that the Score API can detect. This time series has two distinct level changes, and three spikes. The red dots show the time at which the level change is detected, while the black dots show the detected spikes.

Detectors

The anomaly detection API supports detectors in three broad categories. Details on specific input parameters and outputs for each detector can be found in the following table.

DETECTOR CATEGORY	DETECTOR	DESCRIPTION	INPUT PARAMETERS	OUTPUTS
Spike Detectors	TSpike Detector	Detect spikes and dips based on far the values are from first and third quartiles	*tspikedetector.sensitivity:* takes integer value in the range 1-10, default: 3; Higher values will catch more extreme values thus making it less sensitive	TSpike: binary values – '1' if a spike/dip is detected, '0' otherwise

DETECTOR CATEGORY	DETECTOR	DESCRIPTION	INPUT PARAMETERS	OUTPUTS
Spike Detectors	ZSpike Detector	Detect spikes and dips based on how far the datapoints are from their mean	*zspikedetector.sensitivity:* take integer value in the range 1-10, default: 3; Higher values will catch more extreme values making it less sensitive	ZSpike: binary values – '1' if a spike/dip is detected, '0' otherwise
Slow Trend Detector	Slow Trend Detector	Detect slow positive trend as per the set sensitivity	*trenddetector.sensitivity:* threshold on detector score (default: 3.25, 3.25 – 5 is a reasonable range to select from; The higher the less sensitive)	tscore: floating number representing anomaly score on trend
Level Change Detectors	Bidirectional Level Change Detector	Detect both upward and downward level change as per the set sensitivity	*bileveldetector.sensitivity:* threshold on detector score (default: 3.25, 3.25 – 5 is a reasonable range to select from; The higher the less sensitive)	rpscore: floating number representing anomaly score on upward and downward level change

Parameters

More detailed information on these input parameters is listed in the table below:

INPUT PARAMETERS	DESCRIPTION	DEFAULT SETTING	TYPE	VALID RANGE	SUGGESTED RANGE
`detectors.historywindow`	History (in # of data points) used for anomaly score computation	500	integer	10-2000	Time-series dependent
`detectors.spikesdips`	Whether to detect only spikes, only dips, or both	Both	enumerated	Both, Spikes, Dips	Both
`bileveldetector.sensitivity`	Sensitivity for bidirectional level change detector.	3.25	double	None	3.25-5 (Lesser values mean more sensitive)
`trenddetector.sensitivity`	Sensitivity for positive trend detector.	3.25	double	None	3.25-5 (Lesser values mean more sensitive)
`tspikedetector.sensitivity`	Sensitivity for TSpike Detector	3	integer	1-10	3-5 (Lesser values mean more sensitive)

INPUT PARAMETERS	DESCRIPTION	DEFAULT SETTING	TYPE	VALID RANGE	SUGGESTED RANGE
`zspikedetector.sensitivity`	Sensitivity for ZSpike Detector	3	integer	1-10	3-5 (Lesser values mean more sensitive)
`postprocess.tailRows`	Number of the latest data points to be kept in the output results	0	integer	0 (keep all data points), or specify number of points to keep in results	N/A

Output

The API runs all detectors on your time series data and returns anomaly scores and binary spike indicators for each point in time. The table below lists outputs from the API.

OUTPUTS	DESCRIPTION
Time	Timestamps from raw data, or aggregated (and/or) imputed data if aggregation (and/or) missing data imputation is applied
Data	Values from raw data, or aggregated (and/or) imputed data if aggregation (and/or) missing data imputation is applied
TSpike	Binary indicator to indicate whether a spike is detected by TSpike Detector
ZSpike	Binary indicator to indicate whether a spike is detected by ZSpike Detector
rpscore	A floating number representing anomaly score on bidirectional level change
rpalert	1/0 value indicating there is a bidirectional level change anomaly based on the input sensitivity
tscore	A floating number representing anomaly score on positive trend
talert	1/0 value indicating there is a positive trend anomaly based on the input sensitivity

ScoreWithSeasonality API

The ScoreWithSeasonality API is used for running anomaly detection on time series that have seasonal patterns. This API is useful to detect deviations in seasonal patterns. The following figure shows an example of anomalies detected in a seasonal time series. The time series has one spike (the first black dot), two dips (the second black dot and one at the end), and one level change (red dot). Both the dip in the middle of the time series and the level change are only discernable after seasonal components are removed from the series.

Detectors

The detectors in the seasonality endpoint are similar to the ones in the non-seasonality endpoint, but with slightly different parameter names (listed below).

Parameters

More detailed information on these input parameters is listed in the table below:

INPUT PARAMETERS	DESCRIPTION	DEFAULT SETTING	TYPE	VALID RANGE	SUGGESTED RANGE
`preprocess.aggregationInterval`	Aggregation interval in seconds for aggregating input time series	0 (no aggregation is performed)	integer	0: skip aggregation, > 0 otherwise	5 minutes to 1 day, time-series dependent
`preprocess.aggregationFunction`	Function used for aggregating data into the specified AggregationInterval	mean	enumerated	mean, sum, length	N/A
`preprocess.replaceMissingWith`	Values used to impute missing data	lkv (last known value)	enumerated	zero, lkv, mean	N/A
`detectors.historywindow`	History (in # of data points) used for anomaly score computation	500	integer	10-2000	Time-series dependent
`detectors.spikesdips`	Whether to detect only spikes, only dips, or both	Both	enumerated	Both, Spikes, Dips	Both
`bileveldetector.sensitivity`	Sensitivity for bidirectional level change detector.	3.25	double	None	3.25-5 (Lesser values mean more sensitive)
`postrenddetector.sensitivity`	Sensitivity for positive trend detector.	3.25	double	None	3.25-5 (Lesser values mean more sensitive)

INPUT PARAMETERS	DESCRIPTION	DEFAULT SETTING	TYPE	VALID RANGE	SUGGESTED RANGE
`negtrenddetector.sensitivity`	Sensitivity for negative trend detector.	3.25	double	None	3.25-5 (Lesser values mean more sensitive)
`tspikedetector.sensitivity`	Sensitivity for TSpike Detector	3	integer	1-10	3-5 (Lesser values mean more sensitive)
`zspikedetector.sensitivity`	Sensitivity for ZSpike Detector	3	integer	1-10	3-5 (Lesser values mean more sensitive)
`seasonality.enable`	Whether seasonality analysis is to be performed	true	boolean	true, false	Time-series dependent
`seasonality.numSeasonal`	Maximum number of periodic cycles to be detected	1	integer	1, 2	1-2
`seasonality.transform`	Whether seasonal (and) trend components shall be removed before applying anomaly detection	deseason	enumerated	none, deseason, deseasontrend	N/A
`postprocess.tailRows`	Number of the latest data points to be kept in the output results	0	integer	0 (keep all data points), or specify number of points to keep in results	N/A

Output

The API runs all detectors on your time series data and returns anomaly scores and binary spike indicators for each point in time. The table below lists outputs from the API.

OUTPUTS	DESCRIPTION
Time	Timestamps from raw data, or aggregated (and/or) imputed data if aggregation (and/or) missing data imputation is applied
OriginalData	Values from raw data, or aggregated (and/or) imputed data if aggregation (and/or) missing data imputation is applied

OUTPUTS	DESCRIPTION
ProcessedData	Either of the following options: • Seasonally adjusted time series if significant seasonality has been detected and deseason option selected; • seasonally adjusted and detrended time series if significant seasonality has been detected and deseasontrend option selected • otherwise, this option is the same as OriginalData
TSpike	Binary indicator to indicate whether a spike is detected by TSpike Detector
ZSpike	Binary indicator to indicate whether a spike is detected by ZSpike Detector
BiLevelChangeScore	A floating number representing anomaly score on level change
BiLevelChangeAlert	1/0 value indicating there is a level change anomaly based on the input sensitivity
PosTrendScore	A floating number representing anomaly score on positive trend
PosTrendAlert	1/0 value indicating there is a positive trend anomaly based on the input sensitivity
NegTrendScore	A floating number representing anomaly score on negative trend
NegTrendAlert	1/0 value indicating there is a negative trend anomaly based on the input sensitivity

Azure AI for predictive maintenance solutions

11/19/2021 • 42 minutes to read • Edit Online

Summary

Predictive maintenance (**PdM**) is a popular application of predictive analytics that can help businesses in several industries achieve high asset utilization and savings in operational costs. This guide brings together the business and analytical guidelines and best practices to successfully develop and deploy PdM solutions using the Microsoft Azure AI platform technology.

For starters, this guide introduces industry-specific business scenarios and the process of qualifying these scenarios for PdM. The data requirements and modeling techniques to build PdM solutions are also provided. The main content of the guide is on the data science process - including the steps of data preparation, feature engineering, model creation, and model operationalization. To complement these key concepts, this guide lists a set of solution templates to help accelerate PdM application development. The guide also points to useful training resources for the practitioner to learn more about the AI behind the data science.

Data Science guide overview and target audience

The first half of this guide describes typical business problems, the benefits of implementing PdM to address these problems, and lists some common use cases. Business decision makers (BDMs) will benefit from this content. The second half explains the data science behind PdM, and provides a list of PdM solutions built using the principles outlined in this guide. It also provides learning paths and pointers to training material. Technical decision makers (TDMs) will find this content useful.

START WITH ...	IF YOU ARE ...
Business case for predictive maintenance	a business decision maker (BDM) looking to reduce downtime and operational costs, and improve utilization of equipment
Data Science for predictive maintenance	a technical decision maker (TDM) evaluating PdM technologies to understand the unique data processing and AI requirements for predictive maintenance
Solution templates for predictive maintenance	a software architect or AI Developer looking to quickly stand up a demo or a proof-of-concept
Training resources for predictive maintenance	any or all of the above, and want to learn the foundational concepts behind the data science, tools, and techniques.

Prerequisite knowledge

The BDM content does not expect the reader to have any prior data science knowledge. For the TDM content, basic knowledge of statistics and data science is helpful. Knowledge of Azure Data and AI services, Python, R, XML, and JSON is recommended. AI techniques are implemented in Python and R packages. Solution templates are implemented using Azure services, development tools, and SDKs.

Business case for predictive maintenance

Businesses require critical equipment to be running at peak efficiency and utilization to realize their return on capital investments. These assets could range from aircraft engines, turbines, elevators, or industrial chillers - that cost millions - down to everyday appliances like photocopiers, coffee machines, or water coolers.

- By default, most businesses rely on *corrective maintenance*, where parts are replaced as and when they fail. Corrective maintenance ensures parts are used completely (therefore not wasting component life), but costs the business in downtime, labor, and unscheduled maintenance requirements (off hours, or inconvenient locations).
- At the next level, businesses practice *preventive maintenance*, where they determine the useful lifespan for a part, and maintain or replace it before a failure. Preventive maintenance avoids unscheduled and catastrophic failures. But the high costs of scheduled downtime, under-utilization of the component during its useful lifetime, and labor still remain.
- The goal of *predictive maintenance* is to optimize the balance between corrective and preventative maintenance, by enabling *just in time* replacement of components. This approach only replaces those components when they are close to a failure. By extending component lifespans (compared to preventive maintenance) and reducing unscheduled maintenance and labor costs (over corrective maintenance), businesses can gain cost savings and competitive advantages.

Business problems in PdM

Businesses face high operational risk due to unexpected failures and have limited insight into the root cause of problems in complex systems. Some of the key business questions are:

- Detect anomalies in equipment or system performance or functionality.
- Predict whether an asset may fail in the near future.
- Estimate the remaining useful life of an asset.
- Identify the main causes of failure of an asset.
- Identify what maintenance actions need to be done, by when, on an asset.

Typical goal statements from PdM are:

- Reduce operational risk of mission critical equipment.
- Increase rate of return on assets by predicting failures before they occur.
- Control cost of maintenance by enabling just-in-time maintenance operations.
- Lower customer attrition, improve brand image, and lost sales.
- Lower inventory costs by reducing inventory levels by predicting the reorder point.
- Discover patterns connected to various maintenance problems.
- Provide KPIs (key performance indicators) such as health scores for asset conditions.
- Estimate remaining lifespan of assets.
- Recommend timely maintenance activities.
- Enable just in time inventory by estimating order dates for replacement of parts.

These goal statements are the starting points for:

- *data scientists* to analyze and solve specific predictive problems.
- *cloud architects and developers* to put together an end to end solution.

Qualifying problems for predictive maintenance

It is important to emphasize that not all use cases or business problems can be effectively solved by PdM. There are three important qualifying criteria that need to be considered during problem selection:

- The problem has to be predictive in nature; that is, there should be a target or an outcome to predict. The problem should also have a clear path of action to prevent failures when they are detected.
- The problem should have a record of the operational history of the equipment that contains *both good and bad outcomes*. The set of actions taken to mitigate bad outcomes should also be available as part of these records. Error reports, maintenance logs of performance degradation, repair, and replace logs are also

important. In addition, repairs undertaken to improve them, and replacement records are also useful.

- The recorded history should be reflected in *relevant* data that is of *sufficient* enough quality to support the use case. For more information about data relevance and sufficiency, see Data requirements for predictive maintenance.
- Finally, the business should have domain experts who have a clear understanding of the problem. They should be aware of the internal processes and practices to be able to help the analyst understand and interpret the data. They should also be able to make the necessary changes to existing business processes to help collect the right data for the problems, if needed.

Sample PdM use cases

This section focuses on a collection of PdM use cases from several industries such as Aerospace, Utilities, and Transportation. Each section starts with a business problem, and discusses the benefits of PdM, the relevant data surrounding the business problem, and finally the benefits of a PdM solution.

BUSINESS PROBLEM	BENEFITS FROM PDM
Aviation	
Flight delay and cancellations due to mechanical problems. Failures that cannot be repaired in time may cause flights to be canceled, and disrupt scheduling and operations.	PdM solutions can predict the probability of an aircraft being delayed or canceled due to mechanical failures.
Aircraft engine parts failure: Aircraft engine part replacements are among the most common maintenance tasks within the airline industry. Maintenance solutions require careful management of component stock availability, delivery, and planning	Being able to gather intelligence on component reliability leads to substantial reduction on investment costs.
Finance	
ATM failure is a common problem within the banking industry. The problem here is to report the probability that an ATM cash withdrawal transaction gets interrupted due to a paper jam or part failure in the cash dispenser. Based on predictions of transaction failures, ATMs can be serviced proactively to prevent failures from occurring.	Rather than allow the machine to fail midway through a transaction, the desirable alternative is to program the machine to deny service based on the prediction.
Energy	
Wind turbine failures: Wind turbines are the main energy source in environmentally responsible countries/regions, and involve high capital costs. A key component in wind turbines is the generator motor, whose failure renders the turbine ineffective. It is also highly expensive to fix.	Predicting KPIs such as MTTF (mean time to failure) can help the energy companies prevent turbine failures, and ensure minimal downtime. Failure probabilities will inform technicians to monitor turbines that are likely to fail soon, and schedule time-based maintenance regimes. Predictive models provide insights into different factors that contribute to the failure, which helps technicians better understand the root causes of problems.
Circuit breaker failures: Distribution of electricity to homes and businesses requires power lines to be operational at all times to guarantee energy delivery. Circuit breakers help limit or avoid damage to power lines during overloading or adverse weather conditions. The business problem here is to predict circuit breaker failures.	PdM solutions help reduce repair costs and increase the lifespan of equipment such as circuit breakers. They help improve the quality of the power network by reducing unexpected failures and service interruptions.
Transportation and logistics	

BUSINESS PROBLEM	BENEFITS FROM PDM
Elevator door failures: Large elevator companies provide a full stack service for millions of functional elevators around the world. Elevator safety, reliability, and uptime are the main concerns for their customers. These companies track these and various other attributes via sensors, to help them with corrective and preventive maintenance. In an elevator, the most prominent customer problem is malfunctioning elevator doors. The business problem in this case is to provide a knowledge base predictive application that predicts the potential causes of door failures.	Elevators are capital investments for potentially a 20-30 year lifespan. So each potential sale can be highly competitive; hence expectations for service and support are high. Predictive maintenance can provide these companies with an advantage over their competitors in their product and service offerings.
Wheel failures: Wheel failures account for half of all train derailments and cost billions to the global rail industry. Wheel failures also cause rails to deteriorate, sometimes causing the rail to break prematurely. Rail breaks lead to catastrophic events such as derailments. To avoid such instances, railways monitor the performance of wheels and replace them in a preventive manner. The business problem here is the prediction of wheel failures.	Predictive maintenance of wheels will help with just-in-time replacement of wheels
Subway train door failures: A major reason for delays in subway operations is door failures of train cars. The business problem here is to predict train door failures.	Early awareness of a door failure, or the number of days until a door failure, will help the business optimize train door servicing schedules.

The next section gets into the details of how to realize the PdM benefits discussed above.

Data Science for predictive maintenance

This section provides general guidelines of data science principles and practice for PdM. It is intended to help a TDM, solution architect, or a developer understand the prerequisites and process for building end-to-end AI applications for PdM. You can read this section along with a review of the demos and proof-of-concept templates listed in Solution Templates for predictive maintenance. You can then use these principles and best practices to implement your PdM solution in Azure.

> **NOTE**
>
> This guide is NOT intended to teach the reader Data Science. Several helpful sources are provided for further reading in the section for training resources for predictive maintenance. The solution templates listed in the guide demonstrate some of these AI techniques for specific PdM problems.

Data requirements for predictive maintenance

The success of any learning depends on (a) the quality of what is being taught, and (b) the ability of the learner. Predictive models learn patterns from historical data, and predict future outcomes with certain probability based on these observed patterns. A model's predictive accuracy depends on the relevancy, sufficiency, and quality of the training and test data. The new data that is 'scored' using this model should have the same features and schema as the training/test data. The feature characteristics (type, density, distribution, and so on) of new data should match that of the training and test data sets. The focus of this section is on such data requirements.

Relevant data

First, the data has to be *relevant to the problem*. Consider the *wheel failure* use case discussed above - the training data should contain features related to the wheel operations. If the problem was to predict the failure of

the *traction system*, the training data has to encompass all the different components for the traction system. The first case targets a specific component whereas the second case targets the failure of a larger subsystem. The general recommendation is to design prediction systems about specific components rather than larger subsystems, since the latter will have more dispersed data. The domain expert (see Qualifying problems for predictive maintenance) should help in selecting the most relevant subsets of data for the analysis. The relevant data sources are discussed in greater detail in Data preparation for predictive maintenance.

Sufficient data

Two questions are commonly asked with regard to failure history data: (1) "How many failure events are required to train a model?" (2) "How many records is considered as "enough"?" There are no definitive answers, but only rules of thumb. For (1), more the number of failure events, better the model. For (2), and the exact number of failure events depends on the data and the context of the problem being solved. But on the flip side, if a machine fails too often then the business will replace it, which will reduce failure instances. Here again, the guidance from the domain expert is important. However, there are methods to cope with the issue of *rare events*. They are discussed in the section Handling imbalanced data.

Quality data

The quality of the data is critical - each predictor attribute value must be *accurate* in conjunction with the value of the target variable. Data quality is a well-studied area in statistics and data management, and hence out of scope for this guide.

> **NOTE**
>
> There are several resources and enterprise products to deliver quality data. A sample of references is provided below:
>
> - Dasu, T, Johnson, T., Exploratory Data Mining and Data Cleaning, Wiley, 2003.
> - Exploratory Data Analysis, Wikipedia
> - Hellerstein, J, Quantitative Data Cleaning for Large Databases
> - de Jonge, E, van der loo, M, Introduction to Data Cleaning with R

Data preparation for predictive maintenance

Data sources

The relevant data sources for predictive maintenance include, but are not limited to:

- Failure history
- Maintenance/repair history
- Machine operating conditions
- Equipment metadata

Failure history

Failure events are rare in PdM applications. However, when building prediction models, the algorithm needs to learn about a component's normal operational pattern, as well as its failure patterns. So the training data should contain sufficient number of examples from both categories. Maintenance records and parts replacement history are good sources to find failure events. With the help of some domain knowledge, anomalies in the training data can also be defined as failures.

Maintenance/repair history

Maintenance history of an asset contains details about components replaced, repair activities performed etc. These events record degradation patterns. Absence of this crucial information in the training data can lead to misleading model results. Failure history can also be found within maintenance history as special error codes, or order dates for parts. Additional data sources that influence failure patterns should be investigated and provided by domain experts.

Machine operating conditions

Sensor based (or other) streaming data of the equipment in operation is an important data source. A key assumption in PdM is that a machine's health status degrades over time during its routine operation. The data is expected to contain time-varying features that capture this aging pattern, and any anomalies that leads to degradation. The temporal aspect of the data is required for the algorithm to learn the failure and non-failure patterns over time. Based on these data points, the algorithm learns to predict how many more units of time a machine can continue to work before it fails.

Static feature data

Static features are metadata about the equipment. Examples are the equipment make, model, manufactured date, start date of service, location of the system, and other technical specifications.

Examples of relevant data for the sample PdM use cases are tabulated below:

USE CASE	EXAMPLES OF RELEVANT DATA
Flight delay and cancellations	Flight route information in the form of flight legs and page logs. Flight leg data includes routing details such as departure/arrival date, time, airport, layovers etc. Page log includes a series of error and maintenance codes recorded by the ground maintenance personnel.
Aircraft engine parts failure	Data collected from sensors in the aircraft that provide information on the condition of the various parts. Maintenance records help identify when component failures occurred and when they were replaced.
ATM Failure	Sensor readings for each transaction (depositing cash/check) and dispensing of cash. Information on gap measurement between notes, note thickness, note arrival distance, check attributes etc. Maintenance records that provide error codes, repair information, last time the cash dispenser was refilled.
Wind turbine failure	Sensors monitor turbine conditions such as temperature, wind direction, power generated, generator speed etc. Data is gathered from multiple wind turbines from wind farms located in various regions. Typically, each turbine will have multiple sensor readings relaying measurements at a fixed time interval.
Circuit breaker failures	Maintenance logs that include corrective, preventive, and systematic actions. Operational data that includes automatic and manual commands sent to circuit breakers such as for open and close actions. Device metadata such as date of manufacture, location, model, etc. Circuit breaker specifications such as voltage levels, geolocation, ambient conditions.
Elevator door failures	Elevator metadata such as type of elevator, manufactured date, maintenance frequency, building type, and so on. Operational information such as number of door cycles, average door close time. Failure history with causes.
Wheel failures	Sensor data that measures wheel acceleration, braking instances, driving distance, velocity etc. Static information on wheels like manufacturer, manufactured date. Failure data inferred from part order database that track order dates and quantities.

USE CASE	EXAMPLES OF RELEVANT DATA
Subway train door failures	Door opening and closing times, other operational data such as current condition of train doors. Static data would include asset identifier, time, and condition value columns.

Data types

Given the above data sources, the two main data types observed in PdM domain are:

- *Temporal data*: Operational telemetry, machine conditions, work order types, priority codes that will have timestamps at the time of recording. Failure, maintenance/repair, and usage history will also have timestamps associated with each event.
- *Static data*: Machine features and operator features in general are static since they describe the technical specifications of machines or operator attributes. If these features could change over time, they should also have timestamps associated with them.

Predictor and target variables should be preprocessed/transformed into numerical, categorical, and other data types depending on the algorithm being used.

Data preprocessing

As a prerequisite to *feature engineering*, prepare the data from various streams to compose a schema from which it is easy to build features. Visualize the data first as a table of records. Each row in the table represents a training instance, and the columns represent *predictor* features (also called independent attributes or variables). Organize the data such that the last column(s) is the *target* (dependent variable). For each training instance, assign a *label* as the value of this column.

For temporal data, divide the duration of sensor data into time units. Each record should belong to a time unit for an asset, *and should offer distinct information*. Time units are defined based on business needs in multiples of seconds, minutes, hours, days, months, and so on. The time unit *does not have to be the same as the frequency of data collection*. If the frequency is high, the data may not show any significant difference from one unit to the other. For example, assume that ambient temperature was collected every 10 seconds. Using that same interval for training data only inflates the number of examples without providing any additional information. For this case, a better strategy would be to use average the data over 10 minutes, or an hour based on the business justification.

For static data,

- *Maintenance records*: Raw maintenance data has an asset identifier and timestamp with information on maintenance activities that have been performed at a given point in time. Transform maintenance activities into *categorical* columns, where each category descriptor uniquely maps to a specific maintenance action. The schema for maintenance records would include asset identifier, time, and maintenance action.

- *Failure records*: Failures or failure reasons can be recorded as specific error codes or failure events defined by specific business conditions. In cases where the equipment has multiple error codes, the domain expert should help identify the ones that are pertinent to the target variable. Use the remaining error codes or conditions to construct *predictor* features that correlate with these failures. The schema for failure records would include asset identifier, time, failure, or failure reason - if available.

- *Machine and operator metadata*: Merge the machine and operator data into one schema to associate an asset with its operator, along with their respective attributes. The schema for machine conditions would include asset identifier, asset features, operator identifier, and operator features.

Other data preprocessing steps include *handling missing values* and *normalization* of attribute values. A detailed discussion is beyond the scope of this guide - see the next section for some useful references.

With the above preprocessed data sources in place, the final transformation before feature engineering is to join the above tables based on the asset identifier and timestamp. The resulting table would have null values for the failure column when machine is in normal operation. These null values can be imputed by an indicator for normal operation. Use this failure column to create *labels for the predictive model*. For more information, see the section on modeling techniques for predictive maintenance.

Feature engineering

Feature engineering is the first step prior to modeling the data. Its role in the data science process is described here. A *feature* is a predictive attribute for the model - such as temperature, pressure, vibration, and so on. For PdM, feature engineering involves abstracting a machine's health over historical data collected over a sizable duration. In that sense, it is different from its peers such as remote monitoring, anomaly detection, and failure detection.

Time windows

Remote monitoring entails reporting the events that happen as of *points in time*. Anomaly detection models evaluate (score) incoming streams of data to flag anomalies as of points in time. Failure detection classifies failures to be of specific types as they occur points in time. In contrast, PdM involves predicting failures over a *future time period*, based on features that represent machine behavior over *historical time period*. For PdM, feature data from individual points of time are too noisy to be predictive. So the data for each feature needs to be *smoothened* by aggregating data points over time windows.

Lag features

The business requirements define how far the model has to predict into the future. In turn, this duration helps define 'how far back the model has to look' to make these predictions. This 'looking back' period is called the *lag*, and features engineered over this lag period are called *lag features*. This section discusses lag features that can be constructed from data sources with timestamps, and feature creation from static data sources. Lag features are typically *numerical* in nature.

> **IMPORTANT**
>
> The window size is determined via experimentation, and should be finalized with the help of a domain expert. The same caveat holds for the selection and definition of lag features, their aggregations, and the type of windows.

Rolling aggregates

For each record of an asset, a rolling window of size "W" is chosen as the number of units of time to compute the aggregates. Lag features are then computed using the W periods *before the date* of that record. In Figure 1, the blue lines show sensor values recorded for an asset for each unit of time. They denote a rolling average of feature values over a window of size W=3. The rolling average is computed over all records with timestamps in the range t_1 (in orange) to t_2 (in green). The value for W is typically in minutes or hours depending on the nature of the data. But for certain problems, picking a large W (say 12 months) can provide the whole history of an asset until the time of the record.

Figure 1. Rolling aggregate features

Examples of rolling aggregates over a time window are count, average, CUMESUM (cumulative sum) measures, min/max values. In addition, variance, standard deviation, and count of outliers beyond N standard deviations are often used. Examples of aggregates that may be applied for the use cases in this guide are listed below.

- *Flight delay*: count of error codes over the last day/week.
- *Aircraft engine part failure*: rolling means, standard deviation, and sum over the past day, week etc. This metric should be determined along with the business domain expert.
- *ATM failures*: rolling means, median, range, standard deviations, count of outliers beyond three standard deviations, upper and lower CUMESUM.
- *Subway train door failures*: Count of events over past day, week, two weeks etc.
- *Circuit breaker failures*: Failure counts over past week, year, three years etc.

Another useful technique in PdM is to capture trend changes, spikes, and level changes using algorithms that detect anomalies in data.

Tumbling aggregates

For each labeled record of an asset, a window of size $W\text{-}_k$ is defined, where k is the number of windows of size W. Aggregates are then created over *k tumbling windows* $W\text{-}k$, $W\text{-}_{(k-1)}$, ..., $W\text{-}_2$, $W\text{-}_1$ for the periods before a record's timestamp. k can be a small number to capture short-term effects, or a large number to capture long-term degradation patterns. (see Figure 2).

Figure 2. Tumbling aggregate features

For example, lag features for the wind turbines use case may be created with W=1 and k=3. They imply the lag for each of the past three months using top and bottom outliers.

Static features

Technical specifications of the equipment such as date of manufacture, model number, location, are some examples of static features. They are treated as *categorical* variables for modeling. Some examples for the circuit breaker use case are voltage, current, power capacity, transformer type, and power source. For wheel failures,

the type of tire wheels (alloy vs steel) is an example.

The data preparation efforts discussed so far should lead to the data being organized as shown below. Training, test, and validation data should have this logical schema (this example shows time in units of days).

ASSET ID	TIME	<FEATURE COLUMNS>	LABEL
A123	Day 1	. . .	.
A123	Day 2	. . .	.
...	...	. . .	.
B234	Day 1	. . .	.
B234	Day 2	. . .	.
...	...	. . .	.

The last step in feature engineering is the **labeling** of the target variable. This process is dependent on the modeling technique. In turn, the modeling technique depends on the business problem and nature of the available data. Labeling is discussed in the next section.

> **IMPORTANT**
>
> Data preparation and feature engineering are as important as modeling techniques to arrive at successful PdM solutions. The domain expert and the practitioner should invest significant time in arriving at the right features and data for the model. A small sample from many books on feature engineering are listed below:
>
> - Pyle, D. Data Preparation for Data Mining (The Morgan Kaufmann Series in Data Management Systems), 1999
> - Zheng, A., Casari, A. Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists, O'Reilly, 2018.
> - Dong, G. Liu, H. (Editors), Feature Engineering for Machine Learning and Data Analytics (Chapman & Hall/CRC Data Mining and Knowledge Discovery Series), CRC Press, 2018.

Modeling techniques for predictive maintenance

This section discusses the main modeling techniques for PdM problems, along with their specific label construction methods. Notice that a single modeling technique can be used across different industries. The modeling technique is paired to the data science problem, rather than the context of the data at hand.

> **IMPORTANT**
>
> The choice of labels for the failure cases and the labeling strategy should be determined in consultation with the domain expert.

Binary classification

Binary classification is used to *predict the probability that a piece of equipment fails within a future time period* - called the *future horizon period X*. X is determined by the business problem and the data at hand, in consultation with the domain expert. Examples are:

- *minimum lead time* required to replace components, deploy maintenance resources, perform maintenance to avoid a problem that is likely to occur in that period.
- *minimum count of events* that can happen before a problem occurs.

In this technique, two types of training examples are identified. A positive example, *which indicates a failure*, with label = 1. A negative example, which indicates normal operations, with label = 0. The target variable, and hence the label values, are *categorical*. The model should identify each new example as likely to fail or work normally over the next X time units.

Label construction for binary classification

The question here is: "What is the probability that the asset will fail in the next X units of time?" To answer this question, label X records prior to the failure of an asset as "about to fail" (label = 1), and label all other records as being "normal" (label =0). (see Figure 3).

Figure 3. Labeling for binary classification

Examples of labeling strategy for some of the use cases are listed below.

- *Flight delays*: X may be chosen as one day, to predict delays in the next 24 hours. Then all flights that are within 24 hours before failures are labeled as 1.
- *ATM cash dispense failures*: A goal may be to determine failure probability of a transaction in the next one hour. In that case, all transactions that happened within the past hour of the failure are labeled as 1. To predict failure probability over the next N currency notes dispensed, all notes dispensed within the last N notes of a failure are labeled as 1.
- *Circuit breaker failures*: The goal may be to predict the next circuit breaker command failure. In that case, X is chosen to be one future command.
- *Train door failures*: X may be chosen as two days.
- *Wind turbine failures*: X may be chosen as two months.

Regression for predictive maintenance

Regression models are used to *compute the remaining useful life (RUL) of an asset*. RUL is defined as the amount of time that an asset is operational before the next failure occurs. Each training example is a record that belongs to a time unit *nY* for an asset, where *n* is the multiple. The model should calculate the RUL of each new example as a *continuous number*. This number denotes the period of time remaining before the failure.

Label construction for regression

The question here is: "What is the remaining useful life (RUL) of the equipment?" For each record prior to the failure, calculate the label to be the number of units of time remaining before the next failure. In this method, labels are continuous variables. (See Figure 4)

Figure 4. Labeling for regression

For regression, labeling is done with reference to a failure point. Its calculation is not possible without knowing how long the asset has survived before a failure. So in contrast to binary classification, assets without any failures in the data cannot be used for modeling. This issue is best addressed by another statistical technique called Survival Analysis. But potential complications may arise when applying this technique to PdM use cases that involve time-varying data with frequent intervals. For more information on Survival Analysis, see this one-pager.

Multi-class classification for predictive maintenance

Multi-class classification techniques can be used in PdM solutions for two scenarios:

- Predict *two future outcomes*: The first outcome is *a range of time to failure* for an asset. The asset is assigned to one of multiple possible periods of time. The second outcome is the likelihood of failure in a future period due to *one of the multiple root causes*. This prediction enables the maintenance crew to watch for symptoms and plan maintenance schedules.
- Predict *the most likely root cause* of a given failure. This outcome recommends the right set of maintenance actions to fix a failure. A ranked list of root causes and recommended repairs can help technicians prioritize their repair actions after a failure.

Label construction for multi-class classification

The question here is: "What is the probability that an asset will fail in the next *nZ* units of time where *n* is the number of periods?" To answer this question, label nZ records prior to the failure of an asset using buckets of time (3Z, 2Z, Z). Label all other records as "normal" (label = 0). In this method, the target variable holds *categorical* values. (See Figure 5).

Figure 5. Labeling for multi-class classification for failure time prediction

The question here is: "What is the probability that the asset will fail in the next X units of time due to root cause/problem P_i?" where *i* is the number of possible root causes. To answer this question, label X records prior to the failure of an asset as "about to fail due to root cause P_i" (label = P_i). Label all other records as being "normal" (label = 0). In this method also, labels are categorical (See Figure 6).

Figure 6. Labeling for multi-class classification for root cause prediction

The model assigns a failure probability due to each P_i as well as the probability of no failure. These probabilities can be ordered by magnitude to allow prediction of the problems that are most likely to occur in the future.

The question here is: "What maintenance actions do you recommend after a failure?" To answer this question, labeling *does not require a future horizon to be picked*, because the model is not predicting failure in the future. It is just predicting the most likely root cause *once the failure has already happened*.

Training, validation, and testing methods for predictive maintenance

The Team Data Science Process provides a full coverage of the model train-test-validate cycle. This section discusses aspects unique to PdM.

Cross validation

The goal of cross validation is to define a data set to "test" the model in the training phase. This data set is called the *validation set*. This technique helps limit problems like *overfitting* and gives an insight on how the model will generalize to an independent data set. That is, an unknown data set, which could be from a real problem. The training and testing routine for PdM needs to take into account the time varying aspects to better generalize on unseen future data.

Many machine learning algorithms depend on a number of hyperparameters that can change the model performance significantly. The optimal values of these hyperparameters are not computed automatically when training the model. They should be specified by the data scientist. There are several ways of finding good values of hyperparameters.

The most common one is *k-fold cross-validation* that splits the examples randomly into *k* folds. For each set of hyperparameters values, run the learning algorithm *k* times. At each iteration, use the examples in the current fold as a validation set, and the rest of the examples as a training set. Train the algorithm over training examples and compute the performance metrics over validation examples. At the end of this loop, compute the average of *k* performance metrics. For each set of hyperparameter values, choose the ones that have the best average performance. The task of choosing hyperparameters is often experimental in nature.

In PdM problems, data is recorded as a time series of events that come from several data sources. These records may be ordered according to the time of labeling. Hence, if the dataset is split *randomly* into training and validation set, *some of the training examples may be later in time than some of validation examples*. Future performance of hyperparameter values will be estimated based on some data that arrived *before* model was trained. These estimations might be overly optimistic, especially if the time-series is not stationary and evolves over time. As a result, the chosen hyperparameter values might be suboptimal.

The recommended way is to split the examples into training and validation set in a *time-dependent* manner, where all validation examples are later in time than all training examples. For each set of hyperparameter values, train the algorithm over the training data set. Measure the model's performance over the same validation set. Choose hyperparameter values that show the best performance. Hyperparameter values chosen by train/validation split result in better future model performance than with the values chosen randomly by cross-validation.

The final model can be generated by training a learning algorithm over entire training data using the best hyperparameter values.

Testing for model performance

Once a model is built, an estimate of its future performance on new data is required. A good estimate is the performance metric of hyperparameter values computed over the validation set, or an average performance metric computed from cross-validation. These estimations are often overly optimistic. The business might often have some additional guidelines on how they would like to test the model.

The recommended way for PdM is to split the examples into training, validation, and test data sets in a *time-dependent* manner. All test examples should be later in time than all the training and validation examples. After the split, generate the model and measure its performance as described earlier.

When time-series are stationary and easy to predict, both random and time-dependent approaches generate similar estimations of future performance. But when time-series are non-stationary, and/or hard to predict, the time-dependent approach will generate more realistic estimates of future performance.

Time-dependent split

This section describes best practices to implement time-dependent split. A time-dependent two-way split between training and test sets is described below.

Assume a stream of timestamped events such as measurements from various sensors. Define features and labels of training and test examples over time frames that contain multiple events. For example, for binary classification, create features based on past events, and create labels based on future events within "X" units of time in the future (see the sections on feature engineering and modeling techniques). Thus, the labeling time frame of an example comes later than the time frame of its features.

For time-dependent split, pick a *training cutoff time* T_c at which to train a model, with hyperparameters tuned using historical data up to T_c. To prevent leakage of future labels that are beyond T_c into the training data, choose the latest time to label training examples to be X units before T_c. In the example shown in Figure 7, each square represents a record in the data set where features and labels are computed as described above. The figure shows the records that should go into training and testing sets for X=2 and W=3:

Figure 7. Time-dependent split for binary classification

The green squares represent records belonging to the time units that can be used for training. Each training example is generated by considering the past three periods for feature generation, and two future periods for labeling before T_c. When any part of the two future periods is beyond T_c, exclude that example from the training data set because no visibility is assumed beyond T_c.

The black squares represent the records of the final labeled data set that should not be used in the training data set, given the above constraint. These records will also not be used in testing data, since they are before T_c. In addition, their labeling time frames partially depend on the training time frame, which is not ideal. Training and test data should have separate labeling time frames to prevent label information leakage.

The technique discussed so far allows for overlap between training and testing examples that have timestamps

near T_c. A solution to achieve greater separation is to exclude examples that are within W time units of T_c from the test set. But such an aggressive split depends on ample data availability.

Regression models used for predicting RUL are more severely affected by the leakage problem. Using the random split method leads to extreme over-fitting. For regression problems, the split should be such that the records belonging to assets with failures before T_c go into the training set. Records of assets that have failures after the cutoff go into the test set.

Another best practice for splitting data for training and testing is to use a split by asset ID. The split should be such that none of the assets used in the training set are used in testing the model performance. Using this approach, a model has a better chance of providing more realistic results with new assets.

Handling imbalanced data

In classification problems, if there are more examples of one class than of the others, the data set is said to be *imbalanced*. Ideally, enough representatives of each class in the training data are preferred to enable differentiation between different classes. If one class is less than 10% of the data, the data is deemed to be imbalanced. The underrepresented class is called a *minority class*.

Many PdM problems face such imbalanced datasets, where one class is severely underrepresented compared to the other class, or classes. In some situations, the minority class may constitute only 0.001% of the total data points. Class imbalance is not unique to PdM. Other domains where failures and anomalies are rare occurrences face a similar problem, for examples, fraud detection and network intrusion. These failures make up the minority class examples.

With class imbalance in data, performance of most standard learning algorithms is compromised, since they aim to minimize the overall error rate. For a data set with 99% negative and 1% positive examples, a model can be shown to have 99% accuracy by labeling all instances as negative. But the model will mis-classify all positive examples; so even if its accuracy is high, the algorithm is not a useful one. Consequently, conventional evaluation metrics such as *overall accuracy on error rate* are insufficient for imbalanced learning. When faced with imbalanced datasets, other metrics are used for model evaluation:

- Precision
- Recall
- F1 scores
- Cost adjusted ROC (receiver operating characteristics)

For more information about these metrics, see model evaluation.

However, there are some methods that help remedy class imbalance problem. The two major ones are *sampling techniques* and *cost sensitive learning*.

Sampling methods

Imbalanced learning involves the use of sampling methods to modify the training data set to a balanced data set. Sampling methods are not to be applied to the test set. Although there are several sampling techniques, most straight forward ones are *random oversampling* and *under sampling*.

Random oversampling involves selecting a random sample from minority class, replicating these examples, and adding them to training data set. Consequently, the number of examples in minority class is increased, and eventually balance the number of examples of different classes. A drawback of oversampling is that multiple instances of certain examples can cause the classifier to become too specific, leading to over-fitting. The model may show high training accuracy, but its performance on unseen test data may be suboptimal.

Conversely, *random under sampling* is selecting a random sample from a majority class and removing those examples from training data set. However, removing examples from majority class may cause the classifier to miss important concepts pertaining to the majority class. *Hybrid sampling* where minority class is over-sampled and majority class is under-sampled at the same time is another viable approach.

There are many sophisticated sampling techniques. The technique chosen depends on the data properties and results of iterative experiments by the data scientist.

Cost sensitive learning

In PdM, failures that constitute the minority class are of more interest than normal examples. So the focus is mainly on the algorithm's performance on failures. Incorrectly predicting a positive class as a negative class can cost more than vice-versa. This situation is commonly referred as unequal loss or asymmetric cost of mis-classifying elements to different classes. The ideal classifier should deliver high prediction accuracy over the minority class, without compromising on the accuracy for the majority class.

There are multiple ways to achieve this balance. To mitigate the problem of unequal loss, assign a high cost to mis-classification of the minority class, and try to minimize the overall cost. Algorithms like *SVMs (Support Vector Machines)* adopt this method inherently, by allowing cost of positive and negative examples to be specified during training. Similarly, boosting methods such as *boosted decision trees* usually show good performance with imbalanced data.

Model evaluation

Mis-classification is a significant problem for PdM scenarios where the cost of false alarms to the business is high. For instance, a decision to ground an aircraft based on an incorrect prediction of engine failure can disrupt schedules and travel plans. Taking a machine offline from an assembly line can lead to loss of revenue. So model evaluation with the right performance metrics against new test data is critical.

Typical performance metrics used to evaluate PdM models are discussed below:

- Accuracy is the most popular metric used for describing a classifier's performance. But accuracy is sensitive to data distributions, and is an ineffective measure for scenarios with imbalanced data sets. Other metrics are used instead. Tools like confusion matrix are used to compute and reason about accuracy of the model.
- Precision of PdM models relate to the rate of false alarms. Lower precision of the model generally corresponds to a higher rate of false alarms.
- Recall rate denotes how many of the failures in the test set were correctly identified by the model. Higher recall rates mean the model is successful in identifying the true failures.
- F1 score is the harmonic average of precision and recall, with its value ranging between 0 (worst) to 1 (best).

For binary classification,

- Receiver operating curves (ROC) is also a popular metric. In ROC curves, model performance is interpreted based on one fixed operating point on the ROC.
- But for PdM problems, *decile tables* and *lift charts* are more informative. They focus only on the positive class (failures), and provide a more complex picture of the algorithm performance than ROC curves.
  - *Decile tables* are created using test examples in a descending order of failure probabilities. The ordered samples are then grouped into deciles (10% of the samples with highest probability, then 20%, 30%, and so on). The ratio (true positive rate)/(random baseline) for each decile helps estimate the algorithm performance at each decile. The random baseline takes on values 0.1, 0.2, and so on.
  - Lift charts plot the decile true positive rate versus random true positive rate for all deciles. The first deciles are usually the focus of results, since they show the largest gains. First deciles can also be seen as representative for "at risk", when used for PdM.

Model operationalization for predictive maintenance

The benefit the data science exercise is realized only when the trained model is made operational. That is, the model must be deployed into the business systems to make predictions based on new, previously unseen, data. The new data must exactly conform to the *model signature* of the trained model in two ways:

- all the features must be present in every logical instance (say a row in a table) of the new data.

- the new data must be pre-processed, and each of the features engineered, in exactly the same way as the training data.

The above process is stated in many ways in academic and industry literature. But all the following statements mean the same thing:

- *Score new data* using the model
- *Apply the model* to new data
- *Operationalize* the model
- *Deploy* the model
- *Run the model* against new data

As stated earlier, model operationalization for PdM is different from its peers. Scenarios involving anomaly detection and failure detection typically implement *online scoring* (also called *real time scoring*). Here, the model *scores* each incoming record, and returns a prediction. For anomaly detection, the prediction is an indication that an anomaly occurred (Example: One-class SVM). For failure detection, it would be the type or class of failure.

In contrast, PdM involves *batch scoring*. To conform to the model signature, the features in the new data must be engineered in the same manner as the training data. For the large datasets that is typical for new data, features are aggregated over time windows and scored in batch. Batch scoring is typically done in distributed systems like Spark or Azure Batch. There are a couple of alternatives - both suboptimal:

- Streaming data engines support aggregation over windows in memory. So it could be argued that they support online scoring. But these systems are suitable for dense data in narrow windows of time, or sparse elements over wider windows. They may not scale well for the dense data over wider time windows, as seen in PdM scenarios.
- If batch scoring is not available, the solution is to adapt online scoring to handle new data in small batches at a time.

Solution templates for predictive maintenance

The final section of this guide provides a list of PdM solution templates, tutorials, and experiments implemented in Azure. These PdM applications can be deployed into an Azure subscription within minutes in some cases. They can be used as proof-of-concept demos, sandboxes to experiment with alternatives, or accelerators for actual production implementations. These templates are located in the Azure AI Gallery or Azure GitHub. These different samples will be rolled into this solution template over time.

#	TITLE	DESCRIPTION
2	Azure Predictive Maintenance Solution Template	An open-source solution template that demonstrates Azure ML modeling and a complete Azure infrastructure capable of supporting Predictive Maintenance scenarios in the context of IoT remote monitoring.
3	Deep Learning for Predictive Maintenance	Azure Notebook with a demo solution of using LSTM (Long Short-Term Memory) networks (a class of Recurrent Neural Networks) for Predictive Maintenance, with a blog post on this sample.

#	TITLE	DESCRIPTION
4	Azure Predictive Maintenance for Aerospace	One of the first PdM solution templates based on Azure ML v1.0 for aircraft maintenance. This guide originated from this project.
5	Azure AI Toolkit for IoT Edge	AI in the IoT Edge using TensorFlow; toolkit packages deep learning models in Azure IoT Edge-compatible Docker containers and expose those models as REST APIs.
6	Azure IoT Predictive Maintenance	Azure IoT Suite PCS - Preconfigured Solution. Aircraft maintenance PdM template with IoT Suite. Another document and walkthrough related to the same project.
7	Predictive Maintenance template using SQL Server R Services	Demo of remaining useful life scenario based on R services.
8	Predictive Maintenance Modeling Guide	Aircraft maintenance dataset feature engineered using R with experiments and datasets and Azure notebook and experiments in AzureML v1.0

Training resources for predictive maintenance

Microsoft Azure offers learning paths for the foundational concepts behind PdM techniques, besides content and training on general AI concepts and practice.

TRAINING RESOURCE	AVAILABILITY
Microsoft Docs: Data Scientist Role	Public
Microsoft Docs: AI Engineer Role	Public
Microsoft Docs: Data Engineer Role	Public
Microsoft AI School	Public
LinkedIn Learning	Public
Microsoft: Playlists on YouTube for Artificial Intelligence and Analytics	Public
Microsoft AI Show	Public
AI Platform Overview	Public
AI Lab	Public
Microsoft AI	Public

TRAINING RESOURCE	AVAILABILITY
Microsoft Partner Network	Partners

In addition, free MOOCS (massive open online courses) on AI are offered online by academic institutions like Stanford and MIT, and other educational companies.

Solution template for predictive maintenance in aerospace

11/19/2021 • 16 minutes to read • Edit Online

IMPORTANT

This article has been deprecated. The discussion about Predictive Maintenance in Aerospace is still relevant, but for current information, refer to Solution Overview for Business Audiences.

Solution templates are designed to accelerate the process of building an E2E demo. A deployed template provisions your subscription with necessary components and then builds the relationships between them. It also seeds the data pipeline with sample data from a data generator application, which you download and install on your local machine after you deploy the solution template. The data from the generator hydrates the data pipeline and start generating machine learning predictions, which can then be visualized on the Power BI dashboard.

The deployment process guides you through several steps to set up your solution credentials. Make sure you record the credentials such as solution name, username, and password that you provide during the deployment.

The goals of this article are to:

- Describe the reference architecture and components provisioned in your subscription.
- Demonstrate how to replace the sample data with your own data.
- Show how to modify the solution template.

Overview

When you deploy the solution, it activates Azure services including Event Hub, Stream Analytics, HDInsight, Data Factory, and Machine Learning. The architecture diagram shows how the Predictive Maintenance for Aerospace Solution Template is constructed. You can investigate these services in the Azure portal by clicking them in the

solution template diagram created with the solution deployment (except for HDInsight, which is provisioned on demand when the related pipeline activities are required to run and are deleted afterwards). Download a full-size version of the diagram.

The following sections describe the solution parts.

Data source and ingestion

Synthetic data source

For this template, the data source used is generated from a downloaded desktop application that you run locally after successful deployment.

To find the instructions to download and install this application, select the first node, Predictive Maintenance Data Generator, on the solution template diagram. The instructions are found in the Properties bar. This application feeds the Azure Event Hub service with data points, or events, used in the rest of the solution flow. This data source is derived from publicly available data from the NASA data repository using the Turbofan Engine Degradation Simulation Data Set.

The event generation application populates the Azure Event Hub only while it's executing on your computer.

Azure Event Hub

The Azure Event Hub service is the recipient of the input provided by the Synthetic Data Source.

Data preparation and analysis

Azure Stream Analytics

Use Azure Stream Analytics to provide near real-time analytics on the input stream from the Azure Event Hub service. You then publish results onto a Power BI dashboard as well as archive all raw incoming events to the Azure Storage service for later processing by the Azure Data Factory service.

HDInsight custom aggregation

Run Hive scripts (orchestrated by Azure Data Factory) using HDInsight to provide aggregations on the raw events archived using the Azure Stream Analytics resource.

Azure Machine Learning

Make predictions on the remaining useful life (RUL) of a particular aircraft engine using the inputs received with Azure Machine Learning Service (orchestrated by Azure Data Factory). \

Data publishing

Azure SQL Database

Use Azure SQL Database to store the predictions received by the Azure Machine Learning, which are then consumed in the Power BI dashboard.

Data consumption

Power BI

Use Power BI to show a dashboard that contains aggregations and alerts provided by Azure Stream Analytics, as well as RUL predictions stored in Azure SQL Database that were produced using Azure Machine Learning.

How to bring in your own data

This section describes how to bring your own data to Azure, and what areas require changes for the data you bring into this architecture.

It's unlikely that your dataset matches the dataset used by the Turbofan Engine Degradation Simulation Data Set used for this solution template. Understanding your data and the requirements are crucial in how you modify this template to work with your own data.

The following sections discuss the parts of the template that require modifications when a new dataset is introduced.

Azure Event Hub

Azure Event Hub is generic; data can be posted to the hub in either CSV or JSON format. No special processing occurs in the Azure Event Hub, but it's important that you understand the data that's fed into it.

This document does not describe how to ingest your data, but you can easily send events or data to an Azure Event Hub using the Event Hub APIs.

Azure Stream Analytics

Use the Azure Stream Analytics resource to provide near real-time analytics by reading from data streams and outputting data to any number of sources.

For the Predictive Maintenance for Aerospace Solution Template, the Azure Stream Analytics query consists of four sub queries, each query consuming events from the Azure Event Hub service, with outputs to four distinct locations. These outputs consist of three Power BI datasets and one Azure Storage location.

The Azure Stream Analytics query can be found by:

- Connect to the Azure portal
- Locating the Stream Analytics jobs that were generated when the solution was deployed (*for example*, **maintenancesa02asapbi** and **maintenancesa02asablob** for the predictive maintenance solution)
- Selecting
  - ***INPUTS*** to view the query input
  - ***QUERY*** to view the query itself
  - ***OUTPUTS*** to view the different outputs

Information about Azure Stream Analytics query construction can be found in the Stream Analytics Query Reference on MSDN.

In this solution, the queries output three datasets with near real-time analytics information about the incoming data stream to a Power BI dashboard provided as part of this solution template. Because there's implicit knowledge about the incoming data format, these queries must be altered based on your data format.

The query in the second Stream Analytics job **maintenancesa02asablob** simply outputs all Event Hub events to Azure Storage and hence requires no alteration regardless of your data format as the full event information is streamed to storage.

Azure Data Factory

The Azure Data Factory service orchestrates the movement and processing of data. In the Predictive Maintenance for Aerospace Solution Template, the data factory is made up of three pipelines that move and process the data using various technologies. Access your data factory by opening the Data Factory node at the bottom of the solution template diagram created with the deployment of the solution. Errors under your datasets are due to data factory being deployed before the data generator was started. Those errors can be ignored and do not prevent your data factory from functioning.

This section discusses the necessary pipelines and activities contained in the Azure Data Factory. Here is a diagram view of the solution.

Two of the pipelines of this factory contain Hive scripts used to partition and aggregate the data. When noted, the scripts are located in the Azure Storage account created during setup. Their location is: maintenancesascript\\script\\hive\\ (or https://[Your solution name].blob.core.windows.net/maintenancesascript).

Similar to Azure Stream Analytics queries, the Hive scripts have implicit knowledge about the incoming data format and must be altered based on your data format.

AggregateFlightInfoPipeline

This pipeline contains a single activity - an HDInsightHive activity using a HDInsightLinkedService that runs a Hive script to partition the data put in Azure Storage during the Azure Stream Analytics job.

The Hive script for this partitioning task is ***AggregateFlightInfo.hql***

MLScoringPipeline

This pipeline contains several activities whose end result is the scored predictions from the Azure Machine Learning experiment associated with this solution template.

Activities included are:

- HDInsightHive activity using an HDInsightLinkedService that runs a Hive script to perform aggregations and feature engineering necessary for the Azure Machine Learning experiment. The Hive script for this partitioning task is ***PrepareMLInput.hql***.
- Copy activity that moves the results from the HDInsightHive activity to a single Azure Storage blob accessed by the AzureMLBatchScoring activity.
- AzureMLBatchScoring activity calls the Azure Machine Learning experiment, with results put in a single Azure Storage blob.

CopyScoredResultPipeline

This pipeline contains a single activity - a Copy activity that moves the results of the Azure Machine Learning experiment from the ***MLScoringPipeline*** to the Azure SQL Database provisioned as part of the solution template installation.

Azure Machine Learning

The Azure Machine Learning experiment used for this solution template provides the Remaining Useful Life (RUL) of an aircraft engine. The experiment is specific to the data set consumed and requires modification or replacement specific to the data brought in.

Monitor Progress

Once the Data Generator is launched, the pipeline begins to dehydrate, and the different components of your solution start kicking into action following the commands issued by the data factory. There are two ways to monitor the pipeline.

- One of the Stream Analytics jobs writes the raw incoming data to blob storage. If you click on Blob Storage component of your solution from the screen you successfully deployed the solution and then click Open in the right panel, it takes you to the Azure portal. Once there, click on Blobs. In the next panel, you see a list of Containers. Click on **maintenancesadata**. In the next panel is the **rawdata** folder. Inside the rawdata folder are folders with names such as hour=17, and hour=18. The presence of these folders indicates raw data is being generated on your computer and stored in blob storage. You should see csv files with finite sizes in MB in those folders.

- The last step of the pipeline is to write data (for example predictions from machine learning) into SQL Database. You might have to wait a maximum of three hours for the data to appear in SQL Database. One way to monitor how much data is available in your SQL Database is through the Azure portal. On the left panel, locate SQL DATABASES and click it. Then locate your database **pmaintenancedb** and click on it. On the next page at the bottom, click on MANAGE.

  Here, you can click on New Query and query for the number of rows (for example select count(*) from PMResult). As your database grows, the number of rows in the table should increase.

Power BI Dashboard

Set up a Power BI dashboard to visualize your Azure Stream Analytics data (hot path) and batch prediction results from Azure machine learning (cold path).

Set up the cold path dashboard

In the cold path data pipeline, the goal is to get the predictive RUL (remaining useful life) of each aircraft engine once it finishes a flight (cycle). The prediction result is updated every 3 hours for predicting the aircraft engines that have finished a flight during the past 3 hours.

Power BI connects to an Azure SQL Database as its data source, where the prediction results are stored.

Note:

1. On deploying your solution, a prediction will appear in the database within 3 hours. The pbix file that came with the Generator download contains some seed data so that you may create the Power BI dashboard right away.
2. In this step, the prerequisite is to download and install the free software Power BI desktop.

The following steps guide you on how to connect the pbix file to the SQL Database that was spun up at the time of solution deployment containing data (for example, prediction results) for visualization.

1. Get the database credentials.

   You'll need **database server name, database name, user name and password** before moving to next steps. Here are the steps to guide you how to find them.

   - Once **'Azure SQL Database'** on your solution template diagram turns green, click it and then click **'Open'**.
   - You'll see a new browser tab/window that displays the Azure portal page. Click **'Resource groups'** on the left panel.
   - Select the subscription you're using for deploying the solution, and then select **'YourSolutionName_ResourceGroup'**.
   - In the new pop out panel, click the icon to access your database. Your database name is next to this icon (for example, **'pmaintenancedb'**), and the **database server name** is listed under the Server name property and should look similar to **YourSolutionName.database.windows.net**.
   - Your database **username** and **password** are the same as the username and password previously recorded during deployment of the solution.
2. Update the data source of the cold path report file with Power BI Desktop.

   - In the folder where you downloaded and unzipped the Generator file, double-click the **PowerBI\PredictiveMaintenanceAerospace.pbix** file. If you see any warning messages when you open the file, ignore them. On the top of the file, click **'Edit Queries'**.

   - You'll see two tables, **RemainingUsefulLife** and **PMResult**. Select the first table and click next to **'Source'** under **'APPLIED STEPS'** on the right **'Query Settings'** panel. Ignore any warning messages that appear.

   - In the pop out window, replace **'Server'** and **'Database'** with your own server and database names, and then click **'OK'**. For server name, make sure you specify the port 1433 (**YourSolutionName.database.windows.net, 1433**). Leave the Database field as **pmaintenancedb**. Ignore the warning messages that appear on the screen.

   - In the next pop out window, you'll see two options on the left pane (**Windows** and **Database**). Click **'Database'**, fill in your **'Username'** and **'Password'** (the username and password you entered when you first deployed the solution and created an Azure SQL Database). In ***Select which level to apply these settings to***, check database level option. Then click **'Connect'**.

   - Click on the second table **PMResult** then click next to **'Source'** under **'APPLIED STEPS'** on the right **'Query Settings'** panel, and update the server and database names as in the above steps and click OK.

   - Once you're guided back to the previous page, close the window. A message displays - click **Apply**. Lastly, click the **Save** button to save the changes. Your Power BI file has now established connection to the server. If your visualizations are empty, make sure you clear the selections on the visualizations to visualize all the data by clicking the eraser icon on the upper right corner of the legends. Use the refresh button to reflect new data on the visualizations. Initially, you only see the seed data on your visualizations as the data factory is scheduled to refresh every 3 hours. After 3

hours, you will see new predictions reflected in your visualizations when you refresh the data.

3. (Optional) Publish the cold path dashboard to Power BI online. This step needs a Power BI account (or a work or school account).

   - Click **'Publish'** and few seconds later a window appears displaying "Publishing to Power BI Success!" with a green check mark. Click the link below "Open PredictiveMaintenanceAerospace.pbix in Power BI". To find detailed instructions, see Publish from Power BI Desktop.
   - To create a new dashboard: click the **+** sign next to the **Dashboards** section on the left pane. Enter the name "Predictive Maintenance Demo" for this new dashboard.
   - Once you open the report, click to pin all the visualizations to your dashboard. To find detailed instructions, see Pin a tile to a Power BI dashboard from a report. Go to the dashboard page and adjust the size and location of your visualizations and edit their titles. To find detailed instructions on how to edit your tiles, see Edit a tile -- resize, move, rename, pin, delete, add hyperlink. Here is an example dashboard with some cold path visualizations pinned to it. Depending on how long you run your data generator, your numbers on the visualizations may be different.

   - To schedule refresh of the data, hover your mouse over the **PredictiveMaintenanceAerospace** dataset, click ··· and then choose **Schedule Refresh**.

> **NOTE**
>
> If you see a warning message, click **Edit Credentials** and make sure your database credentials are the same as those described in step 1.
>
>
>

   - Expand the **Schedule Refresh** section. Turn on "keep your data up-to-date".
   - Schedule the refresh based on your needs. To find more information, see Data refresh in Power BI.

Setup hot path dashboard

The following steps guide you how to visualize data output from Stream Analytics jobs that were generated at

the time of solution deployment. A Power BI online account is required to perform the following steps. If you don't have an account, you can create one.

1. Add Power BI output in Azure Stream Analytics (ASA).

   - You must follow the instructions in Azure Stream Analytics & Power BI: An analytics dashboard for real-time visibility of streaming data to set up the output of your Azure Stream Analytics job as your Power BI dashboard.
   - The ASA query has three outputs that are **aircraftmonitor**, **aircraftalert**, and **flightsbyhour**. You can view the query by clicking on query tab. Corresponding to each of these tables, you need to add an output to ASA. When you add the first output (**aircraftmonitor**) make sure the **Output Alias**, **Dataset Name** and **Table Name** are the same (**aircraftmonitor**). Repeat the steps to add outputs for **aircraftalert**, and **flightsbyhour**. Once you have added all three output tables and started the ASA job, you should get a confirmation message ("Starting Stream Analytics job maintenancesa02asapbi succeeded").
2. Log in to Power BI online

   - On the left panel Datasets section in My Workspace, the ***DATASET*** names **aircraftmonitor**, **aircraftalert**, and **flightsbyhour** should appear. This is the streaming data you pushed from Azure Stream Analytics in the previous step. The dataset **flightsbyhour** may not show up at the same time as the other two datasets due to the nature of the SQL query behind it. However, it should show up after an hour.
   - Make sure the ***Visualizations*** pane is open and is shown on the right side of the screen.
3. Once you have the data flowing into Power BI, you can start visualizing the streaming data. Below is an example dashboard with some hot path visualizations pinned to it. You can create other dashboard tiles based on appropriate datasets. Depending on how long you run your data generator, your numbers on the visualizations may be different.

4. Here are some steps to create one of the tiles above – the "Fleet View of Sensor 11 vs. Threshold 48.26" tile:

   - Click dataset **aircraftmonitor** on the left panel Datasets section.
   - Click the **Line Chart** icon.
   - Click **Processed** in the **Fields** pane so that it shows under "Axis" in the **Visualizations** pane.
   - Click "s11" and "s11_alert" so that they both appear under "Values". Click the small arrow next to **s11**

and **s11_alert**, change "Sum" to "Average".

- Click **SAVE** on the top and name the report "aircraftmonitor." The report named "aircraftmonitor" is shown in the **Reports** section in the **Navigator** pane on the left.
- Click the **Pin Visual** icon on the top-right corner of this line chart. A "Pin to Dashboard" window may show up for you to choose a dashboard. Select "Predictive Maintenance Demo," then click "Pin."
- Hover the mouse over this tile on the dashboard, click the "edit" icon on the top-right corner to change its title to "Fleet View of Sensor 11 vs. Threshold 48.26" and subtitle to "Average across fleet over time."

Delete your solution

Ensure that you stop the data generator when not actively using the solution as running the data generator will incur higher costs. Delete the solution if you are not using it. Deleting your solution deletes all the components provisioned in your subscription when you deployed the solution. To delete the solution, click your solution name in the left panel of the solution template, and then click **Delete**.

Cost estimation tools

The following two tools are available to help you better understand the total costs involved in running the Predictive Maintenance for Aerospace Solution Template in your subscription:

- Microsoft Azure Cost Estimator Tool (online)
- Microsoft Azure Cost Estimator Tool (desktop)

Baseball decision analysis with ML.NET and Blazor

11/19/2021 • 5 minutes to read • Edit Online

This scenario describes a web application, called the Baseball Machine Learning Workbench, which provides an interface for non-technical users to use artificial intelligence (AI) and machine learning (ML) to perform decision analysis techniques in order to rapidly gain insights and make informed predictions.

This solution uses historical baseball data to generate National Baseball Hall of Fame insights. Machine intelligence powers the what-if analysis, decision thresholding, and improvements over traditional rule-based systems. User-friendly interface controls set adjustable parameters and surface the results in real time, with clear visual cues to highlight positive or negative outcomes.

The architecture provides rapid results by using in-memory models and rapid two-way communication between the user and server. Delta rendering pushes down only modified content to the web browser, updating the display without having to reload the entire page. This solution can scale to tens of in-memory models serving hundreds of concurrent sessions in real time.

The following article explains the architecture of the Baseball Machine Learning Workbench, where to get the source code for it, and how to deploy it. You can also view a live demo of this solution.

Relevant use cases

Consider this scenario for the following use cases:

- Decision analysis system replacement with AI and ML
- AI-assisted decision support systems or decision management systems
- Fantasy baseball
- Financial forecasting
- Business goal and objective modeling (budgeting and project planning, for example)

Architecture

The processing sequence in this solution flows as follows:

1. The user accesses the workbench application with any browser. They choose which analysis method to employ and then which player to analyze.
2. SignalR brokers two-way communication with the server in real time.
3. Azure App Service hosts the application, including AI logic with the machine learning models.
4. One of three different decision analysis mechanisms is utilized, depending on which mode the user has selected.
5. Historical data is analyzed using the designated set of rules or ML models in ML.NET, operating in-memory for very quick inference.
6. Blazor Server surfaces the results to the end user's browser, updating only the portions of the interface that have changed, and transmits back to the user via SignalR.
7. Azure Application Insights is optionally used to monitor performance and instrumentation resources as needed.

Components

The following assets and technologies were used to craft the Baseball Machine Learning Workbench:

- Azure App Service enables you to build and host web applications in the programming language of your choice without managing infrastructure.
- .NET Core 3.1 is an open-source, cross-platform, general-purpose development framework that runs on Windows, Linux, and macOS platforms.
- ML.NET is a cross-platform, open-source framework for creating machine learning and artificial intelligence models using .NET. It is used in the inference sections of this application.
- Blazor Server lets you build interactive web UIs using C# instead of JavaScript, as in this solution. In this architecture, Blazor renders the UI on the server and pushes HTML and other changes out to the browser.
- SignalR provides asynchronous communication between the browser and the server (including Blazor). It

handles event updates, UI updates, and any processing done on the server (such as model inference).

- Visual Studio 2019 is the software programming environment used for this project. This architecture uses cross-platform components, so either the Windows or Mac version can be used.
- Azure Application Insights (a feature of Azure Monitor) can be used for performance monitoring and analytics and to drive autoscaling.

Considerations

Scalability

This solution uses the prediction engine functionality in ML.NET to scale the model response times. Object pooling allows the ML.NET models to be accessed by multiple requests in a thread-safe manner. Learn more about ML.NET object pooling in Deploy a model in an ASP.NET Core Web API.

Azure App Service is used for hosting the workbench in the cloud. With App Service you can automatically scale the number of instances that run your app, letting you keep up with customer demand. For more information on autoscale, refer to Autoscaling best practices in the Azure Architecture Center.

In Blazor Server, the state of many components might be maintained concurrently by the server. Because of this, memory exhaustion is a concern that must be addressed. For guidance on how to author a Blazor Server app to help ensure the best use of server memory, consult Threat mitigation guidance for ASP.NET Core Blazor Server. Applying these best practices allows a server-side Blazor application to scale to thousands of concurrent users—even on relatively small server hosts.

General guidance on designing scalable solutions is provided in the Azure Architecture Center's Performance efficiency checklist.

Resiliency and support

Use .NET Core 3.1.x because it is a Long Term Support (LTS) release. Although Blazor Server is also available in .NET Core 3.0, that is not an LTS release and thus continuing compatibility with future component updates is not assured. Learn more about the .NET Core Support Policy.

Deploy this solution

All of the source code for this scenario is available in the Baseball Machine Learning Workbench repository. This solution is open source and provided with an MIT License.

Prerequisites

For online deployments, you must have an existing Azure account. If you need one, create a free Azure account before you begin.

For deployment as an Azure application instance, you need the Visual Studio 2019 IDE and you must have Git installed locally.

The historical baseball data used for the analysis and machine learning models comes from Sean Lahman's Baseball Database, which is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. The Major League Baseball data itself is public domain.

Deployment to Azure

1. Make sure you have your Azure subscription information handy.
2. Start by cloning the workbench GitHub repository:

```
git clone https://github.com/bartczernicki/MachineLearning-BaseballPrediction-BlazorApp.git
```

3. Follow the instructions provided in the GETSTARTED.md file.

Alternative: Docker container

This application is also available as a complete, ready-to-run Docker container downloadable from Docker Hub.

The container can be run locally (offline) in your own environment. It can also be deployed online in an Azure Container Instance. Instructions for getting started with either use case are provided in the main GitHub repo's Get Started documentation:

- Run the Docker Container locally in your own environment
- Publish Docker Container to the Azure Cloud using Azure Container Instances

Related resources

- View a live demo of this solution
- Baseball HOF prediction using R mlr and DALEX packages is a GitHub repo using R and cutting edge "black box" model techniques to explain ML.NET models related to this workload
- Blazor documentation
- ML.NET documentation
- ASP.NET Core Blazor hosting models
- MLOps (DevOps for Machine Learning) helps data science teams deliver innovation faster, increasing the pace of ML model development
- Learn about the National Baseball Hall of Fame voting process and rules
- XAI Stories: Case Studies for Explainable Artificial Intelligence (Warsaw University of Technology and University of Warsaw, 2020)

Choose an analytical data store in Azure

11/19/2021 • 5 minutes to read • Edit Online

In a big data architecture, there is often a need for an analytical data store that serves processed data in a structured format that can be queried using analytical tools. Analytical data stores that support querying of both hot-path and cold-path data are collectively referred to as the serving layer, or data serving storage.

The serving layer deals with processed data from both the hot path and cold path. In the lambda architecture, the serving layer is subdivided into a *speed serving* layer, which stores data that has been processed incrementally, and a *batch serving* layer, which contains the batch-processed output. The serving layer requires strong support for random reads with low latency. Data storage for the speed layer should also support random writes, because batch loading data into this store would introduce undesired delays. On the other hand, data storage for the batch layer does not need to support random writes, but batch writes instead.

There is no single best data management choice for all data storage tasks. Different data management solutions are optimized for different tasks. Most real-world cloud apps and big data processes have a variety of data storage requirements and often use a combination of data storage solutions.

What are your options when choosing an analytical data store?

There are several options for data serving storage in Azure, depending on your needs:

- Azure Synapse Analytics
- Azure Synapse Spark pools
- Azure Databricks
- Azure Data Explorer
- Azure SQL Database
- SQL Server in Azure VM
- HBase/Phoenix on HDInsight
- Hive LLAP on HDInsight
- Azure Analysis Services
- Azure Cosmos DB

These options provide various database models that are optimized for different types of tasks:

- Key/value databases hold a single serialized object for each key value. They're good for storing large volumes of data where you want to get one item for a given key value and you don't have to query based on other properties of the item.
- Document databases are key/value databases in which the values are *documents*. A "document" in this context is a collection of named fields and values. The database typically stores the data in a format such as XML, YAML, JSON, or BSON, but may use plain text. Document databases can query on non-key fields and define secondary indexes to make querying more efficient. This makes a document database more suitable for applications that need to retrieve data based on criteria more complex than the value of the document key. For example, you could query on fields such as product ID, customer ID, or customer name.
- Column-family databases are key/value data stores that structure data storage into collections of related columns called column families. For example, a census database might have one group of columns for a person's name (first, middle, last), one group for the person's address, and one group for the person's profile information (data of birth, gender). The database can store each column family in a separate partition, while keeping all of the data for one person related to the same key. An application can read a single column family

without reading through all of the data for an entity.

- Graph databases store information as a collection of objects and relationships. A graph database can efficiently perform queries that traverse the network of objects and the relationships between them. For example, the objects might be employees in a human resources database, and you might want to facilitate queries such as "find all employees who directly or indirectly work for Scott."
- Telemetry and time series databases are an append-only collection of objects. Telemetry databases efficiently index data in a variety of column stores and in-memory structures, making them the optimal choice for storing and analyzing vast quantities of telemetry and time series data.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need serving storage that can serve as a hot path for your data? If yes, narrow your options to those that are optimized for a speed serving layer.
- Do you need massively parallel processing (MPP) support, where queries are automatically distributed across several processes or nodes? If yes, select an option that supports query scale out.
- Do you prefer to use a relational data store? If so, narrow your options to those with a relational database model. However, note that some non-relational stores support SQL syntax for querying, and tools such as PolyBase can be used to query non-relational data stores.
- Do you collect time series data? Do you use append-only data?

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Is managed service	Yes	Yes	Yes	Yes	Yes [1]	Yes [1]	Yes	Yes
Primary database model	Relational (columnar format when using columnstore indexes)	Relational tables with columnar storage	Wide column store	Relational (column store), telemetry, and time series store	Wide column store	Hive/In-Memory	Tabular semantic models	Document store, graph, key-value store, wide column store
SQL language support	Yes	Yes	Yes	Yes	Yes (using Phoenix JDBC driver)	Yes	No	Yes

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Optimized for speed serving layer	Yes [2]	Yes [3]	Yes	Yes	Yes	Yes	No	Yes

[1] With manual configuration and scaling.

[2] Using memory-optimized tables and hash or nonclustered indexes.

[3] Supported as an Azure Stream Analytics output.

Scalability capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Redundant regional servers for high availability	Yes	No	No	Yes	Yes	No	No	Yes
Supports query scale out	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic scalability (scale up)	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Supports in-memory caching of data	Yes	Yes	Yes	Yes	No	Yes	Yes	No

Security capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Authentication	SQL / Azure Active Directory (Azure AD)	SQL / Azure AD	Azure AD	local / Azure AD [1]	local / Azure AD [1]	Azure AD	database users / Azure AD via access control (IAM)

CAPABILITY	SQL DATABASE	AZURE SYNAPSE	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Data encryption at rest	Yes [2]	Yes [2]	Yes	Yes [1]	Yes [1]	Yes	Yes
Row-level security	Yes	Yes [3]	No	Yes [1]	Yes [1]	Yes	No
Supports firewalls	Yes	Yes	Yes	Yes [4]	Yes [4]	Yes	Yes
Dynamic data masking	Yes	Yes	Yes	Yes [1]	Yes	No	No

[1] Requires using a domain-joined HDInsight cluster.

[2] Requires using transparent data encryption (TDE) to encrypt and decrypt your data at rest.

[3] Filter predicates only. See Row-Level Security

[4] When used within an Azure Virtual Network. See Extend Azure HDInsight using an Azure Virtual Network.

Choose a data analytics technology in Azure

11/19/2021 • 4 minutes to read • Edit Online

The goal of most big data solutions is to provide insights into the data through analysis and reporting. This can include preconfigured reports and visualizations, or interactive data exploration.

What are your options when choosing a data analytics technology?

There are several options for analysis, visualizations, and reporting in Azure, depending on your needs:

- Power BI
- Jupyter Notebooks
- Zeppelin Notebooks
- Microsoft Azure Notebooks

Power BI

Power BI is a suite of business analytics tools. It can connect to hundreds of data sources, and can be used for ad hoc analysis. See this list of the currently available data sources. Use Power BI Embedded to integrate Power BI within your own applications without requiring any additional licensing.

Organizations can use Power BI to produce reports and publish them to the organization. Everyone can create personalized dashboards, with governance and security built in. Power BI uses Azure Active Directory (Azure AD) to authenticate users who log in to the Power BI service, and uses the Power BI login credentials whenever a user attempts to access resources that require authentication.

Jupyter Notebooks

Jupyter Notebooks provide a browser-based shell that lets data scientists create *notebook* files that contain Python, Scala, or R code and markdown text, making it an effective way to collaborate by sharing and documenting code and results in a single document.

Most varieties of HDInsight clusters, such as Spark or Hadoop, come preconfigured with Jupyter notebooks for interacting with data and submitting jobs for processing. Depending on the type of HDInsight cluster you are using, one or more kernels will be provided for interpreting and running your code. For example, Spark clusters on HDInsight provide Spark-related kernels that you can select from to execute Python or Scala code using the Spark engine.

Jupyter notebooks provide a great environment for analyzing, visualizing, and processing your data prior to building more advanced visualizations with a BI/reporting tool like Power BI.

Zeppelin Notebooks

Zeppelin Notebooks are another option for a browser-based shell, similar to Jupyter in functionality. Some HDInsight clusters come preconfigured with Zeppelin notebooks. However, if you are using an HDInsight Interactive Query (Hive LLAP) cluster, Zeppelin is currently your only choice of notebook that you can use to run interactive Hive queries. Also, if you are using a domain-joined HDInsight cluster, Zeppelin notebooks are the only type that enables you to assign different user logins to control access to notebooks and the underlying Hive tables.

Microsoft Azure Notebooks

Azure Notebooks is an online Jupyter Notebooks-based service that enables data scientists to create, run, and share Jupyter Notebooks in cloud-based libraries. Azure Notebooks provides execution environments for Python 2, Python 3, F#, and R, and provides several charting libraries for visualizing your data, such as ggplot,

matplotlib, bokeh, and seaborn.

Unlike Jupyter notebooks running on an HDInsight cluster, which are connected to the cluster's default storage account, Azure Notebooks does not provide any data. You must load data in a variety of ways, such downloading data from an online source, interacting with Azure Blobs or Table Storage, connecting to a SQL database, or loading data with the Copy Wizard for Azure Data Factory.

Key benefits:

- Free service—no Azure subscription required.
- No need to install Jupyter and the supporting R or Python distributions locally—just use a browser.
- Manage your own online libraries and access them from any device.
- Share your notebooks with collaborators.

Considerations:

- You will be unable to access your notebooks when offline.
- Limited processing capabilities of the free notebook service may not be enough to train large or complex models.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need to connect to numerous data sources, providing a centralized place to create reports for data spread throughout your domain? If so, choose an option that allows you to connect to 100s of data sources.
- Do you want to embed dynamic visualizations in an external website or application? If so, choose an option that provides embedding capabilities.
- Do you want to design your visualizations and reports while offline? If yes, choose an option with offline capabilities.
- Do you need heavy processing power to train large or complex AI models or work with very large data sets? If yes, choose an option that can connect to a big data cluster.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	POWER BI	JUPYTER NOTEBOOKS	ZEPPELIN NOTEBOOKS	MICROSOFT AZURE NOTEBOOKS
Connect to big data cluster for advanced processing	Yes	Yes	Yes	No
Managed service	Yes	Yes [1]	Yes [1]	Yes
Connect to 100s of data sources	Yes	No	No	No
Offline capabilities	Yes [2]	No	No	No

CAPABILITY	POWER BI	JUPYTER NOTEBOOKS	ZEPPELIN NOTEBOOKS	MICROSOFT AZURE NOTEBOOKS
Embedding capabilities	Yes	No	No	No
Automatic data refresh	Yes	No	No	No
Access to numerous open source packages	No	Yes [3]	Yes [3]	Yes [4]
Data transformation/cleansing options	Power Query, R	40 languages, including Python, R, Julia, and Scala	20+ interpreters, including Python, JDBC, and R	Python, F#, R
Pricing	Free for Power BI Desktop (authoring), see pricing for hosting options	Free	Free	Free
Multiuser collaboration	Yes	Yes (through sharing or with a multiuser server like JupyterHub)	Yes	Yes (through sharing)

[1] When used as part of a managed HDInsight cluster.

[2] With the use of Power BI Desktop.

[2] You can search the Maven repository for community-contributed packages.

[3] Python packages can be installed using either pip or conda. R packages can be installed from CRAN or GitHub. Packages in F# can be installed via nuget.org using the Paket dependency manager.

Choose a batch processing technology in Azure

11/19/2021 • 3 minutes to read • Edit Online

Big data solutions often use long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. Usually these jobs involve reading source files from scalable storage (like HDFS, Azure Data Lake Store, and Azure Storage), processing them, and writing the output to new files in scalable storage.

The key requirement of such batch processing engines is the ability to scale out computations, in order to handle a large volume of data. Unlike real-time processing, however, batch processing is expected to have latencies (the time between data ingestion and computing a result) that measure in minutes to hours.

Technology choices for batch processing

Azure Synapse Analytics

Azure Synapse is a distributed system designed to perform analytics on large data. It supports massive parallel processing (MPP), which makes it suitable for running high-performance analytics. Consider Azure Synapse when you have large amounts of data (more than 1 TB) and are running an analytics workload that will benefit from parallelism.

Azure Data Lake Analytics

Data Lake Analytics is an on-demand analytics job service. It is optimized for distributed processing of very large data sets stored in Azure Data Lake Store.

- Languages: U-SQL (including Python, R, and C# extensions).
- Integrates with Azure Data Lake Store, Azure Storage blobs, Azure SQL Database, and Azure Synapse.
- Pricing model is per-job.

HDInsight

HDInsight is a managed Hadoop service. Use it deploy and manage Hadoop clusters in Azure. For batch processing, you can use Spark, Hive, Hive LLAP, MapReduce.

- Languages: R, Python, Java, Scala, SQL
- Kerberos authentication with Active Directory, Apache Ranger based access control
- Gives you full control of the Hadoop cluster

Azure Databricks

Azure Databricks is an Apache Spark-based analytics platform. You can think of it as "Spark as a service." It's the easiest way to use Spark on the Azure platform.

- Languages: R, Python, Java, Scala, Spark SQL
- Fast cluster start times, autotermination, autoscaling.
- Manages the Spark cluster for you.
- Built-in integration with Azure Blob Storage, Azure Data Lake Storage (ADLS), Azure Synapse, and other services. See Data Sources.
- User authentication with Azure Active Directory.
- Web-based notebooks for collaboration and data exploration.
- Supports GPU-enabled clusters

Azure Distributed Data Engineering Toolkit

The Distributed Data Engineering Toolkit (AZTK) is a tool for provisioning on-demand Spark on Docker clusters

in Azure.

AZTK is not an Azure service. Rather, it's a client-side tool with a CLI and Python SDK interface, that's built on Azure Batch. This option gives you the most control over the infrastructure when deploying a Spark cluster.

- Bring your own Docker image.
- Use low-priority VMs for an 80% discount.
- Mixed mode clusters that use both low-priority and dedicated VMs.
- Built in support for Azure Blob Storage and Azure Data Lake connection.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Do you want to author batch processing logic declaratively or imperatively?
- Will you perform batch processing in bursts? If yes, consider options that let you auto-terminate the cluster or whose pricing model is per batch job.
- Do you need to query relational data stores along with your batch processing, for example to look up reference data? If yes, consider the options that enable querying of external relational stores.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE DATA LAKE ANALYTICS	AZURE SYNAPSE	HDINSIGHT	AZURE DATABRICKS
Is managed service	Yes	Yes	Yes [1]	Yes
Relational data store	Yes	Yes	No	No
Pricing model	Per batch job	By cluster hour	By cluster hour	Databricks Unit[2] + cluster hour

[1] With manual configuration.

[2] A Databricks Unit (DBU) is a unit of processing capability per hour.

Capabilities

CAPABILITY	AZURE DATA LAKE ANALYTICS	AZURE SYNAPSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP	AZURE DATABRICKS
Autoscaling	No	No	Yes	Yes	Yes	Yes
Scale-out granularity	Per job	Per cluster	Per cluster	Per cluster	Per cluster	Per cluster
In-memory caching of data	No	Yes	Yes	No	Yes	Yes

CAPABILITY	AZURE DATA LAKE ANALYTICS	AZURE SYNAPSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP	AZURE DATABRICKS
Query from external relational stores	Yes	No	Yes	No	No	Yes
Authentication	Azure AD	SQL / Azure AD	No	Azure AD[1]	Azure AD[1]	Azure AD
Auditing	Yes	Yes	No	Yes [1]	Yes [1]	Yes
Row-level security	No	Yes[2]	No	Yes [1]	Yes [1]	No
Supports firewalls	Yes	Yes	Yes	Yes [3]	Yes [3]	No
Dynamic data masking	No	Yes	No	Yes [1]	Yes [1]	No

[1] Requires using a domain-joined HDInsight cluster.

[2] Filter predicates only. See Row-Level Security

[3] Supported when used within an Azure Virtual Network.

Next steps

- Analytics architecture design
- Choose an analytical data store in Azure
- Choose a data analytics technology in Azure
- Analytics end-to-end with Azure Synapse

High Performance Computing (HPC) on Azure

11/19/2021 • 7 minutes to read • Edit Online

Introduction to HPC

High Performance Computing (HPC), also called "Big Compute", uses a large number of CPU or GPU-based computers to solve complex mathematical tasks.

Many industries use HPC to solve some of their most difficult problems. These include workloads such as:

- Genomics
- Oil and gas simulations
- Finance
- Semiconductor design
- Engineering
- Weather modeling

How is HPC different on the cloud?

One of the primary differences between an on-premises HPC system and one in the cloud is the ability for resources to dynamically be added and removed as they're needed. Dynamic scaling removes compute capacity as a bottleneck and instead allow customers to right size their infrastructure for the requirements of their jobs.

The following articles provide more detail about this dynamic scaling capability.

- Big Compute Architecture Style
- Autoscaling best practices

Implementation checklist

As you're looking to implement your own HPC solution on Azure, ensure you're reviewed the following topics:

- Choose the appropriate architecture based on your requirements
- Know which compute options is right for your workload
- Identify the right storage solution that meets your needs
- Decide how you're going to manage all your resources
- Optimize your application for the cloud
- Secure your Infrastructure

Infrastructure

There are a number of infrastructure components necessary to build an HPC system. Compute, Storage, and Networking provide the underlying components, no matter how you choose to manage your HPC workloads.

Example HPC architectures

There are a number of different ways to design and implement your HPC architecture on Azure. HPC applications can scale to thousands of compute cores, extend on-premises clusters, or run as a 100% cloud-native solution.

The following scenarios outline a few of the common ways HPC solutions are built.

Computer-aided engineering services on Azure

Provide a software-as-a-service (SaaS) platform for computer-aided engineering (CAE) on Azure.

Computational fluid dynamics (CFD) simulations on Azure

Execute computational fluid dynamics (CFD) simulations on Azure.

- **3D video rendering on Azure**

  Run native HPC workloads in Azure using the Azure Batch service

Compute

Azure offers a range of sizes that are optimized for both CPU & GPU intensive workloads.

CPU-based virtual machines

- Linux VMs
- Windows VMs VMs

GPU-enabled virtual machines

N-series VMs feature NVIDIA GPUs designed for compute-intensive or graphics-intensive applications including artificial intelligence (AI) learning and visualization.

- Linux VMs
- Windows VMs

Storage

Large-scale Batch and HPC workloads have demands for data storage and access that exceed the capabilities of traditional cloud file systems. There are a number of solutions to manage both the speed and capacity needs of HPC applications on Azure

- Avere vFXT for faster, more accessible data storage for high-performance computing at the edge
- Azure NetApp Files
- BeeGFS
- Storage Optimized Virtual Machines
- Blob, table, and queue storage
- Azure SMB File storage

For more information comparing Lustre, GlusterFS, and BeeGFS on Azure, review the Parallel Files Systems on Azure e-book and the Lustre on Azure blog.

Networking

H16r, H16mr, A8, and A9 VMs can connect to a high throughput back-end RDMA network. This network can improve the performance of tightly coupled parallel applications running under Microsoft MPI or Intel MPI.

- RDMA Capable Instances

- Virtual Network
- ExpressRoute

Management

Do-it-yourself

Building an HPC system from scratch on Azure offers a significant amount of flexibility, but is often very maintenance intensive.

1. Set up your own cluster environment in Azure virtual machines or virtual machine scale sets.
2. Use Azure Resource Manager templates to deploy leading workload managers, infrastructure, and applications.
3. Choose HPC and GPU VM sizes that include specialized hardware and network connections for MPI or GPU workloads.
4. Add high performance storage for I/O-intensive workloads.

Hybrid and cloud Bursting

If you have an existing on-premises HPC system that you'd like to connect to Azure, there are a number of resources to help get you started.

First, review the Options for connecting an on-premises network to Azure article in the documentation. From there, you may want information on these connectivity options:

- **Connect an on-premises network to Azure using a VPN gateway**

  This reference architecture shows how to extend an on-premises network to Azure, using a site-to-site virtual private network (VPN).

- **Connect an on-premises network to Azure using ExpressRoute**

  ExpressRoute connections use a private, dedicated connection through a third-party connectivity provider. The private connection extends your on-premises network into Azure.

- **Connect an on-premises network to Azure using ExpressRoute with VPN failover**

  Implement a highly available and secure site-to-site network architecture that spans an Azure virtual network and an on-premises network connected using ExpressRoute with VPN gateway failover.

Once network connectivity is securely established, you can start using cloud compute resources on-demand with the bursting capabilities of your existing workload manager.

Marketplace solutions

There are a number of workload managers offered in the Azure Marketplace.

- RogueWave CentOS-based HPC
- SUSE Linux Enterprise Server for HPC
- TIBCO Datasynapse GridServer
- Azure Data Science VM for Windows and Linux

- D3View
- UberCloud

Azure Batch

Azure Batch is a platform service for running large-scale parallel and high-performance computing (HPC) applications efficiently in the cloud. Azure Batch schedules compute-intensive work to run on a managed pool of virtual machines, and can automatically scale compute resources to meet the needs of your jobs.

SaaS providers or developers can use the Batch SDKs and tools to integrate HPC applications or container workloads with Azure, stage data to Azure, and build job execution pipelines.

Azure CycleCloud

Azure CycleCloud Provides the simplest way to manage HPC workloads using any scheduler (like Slurm, Grid Engine, HPC Pack, HTCondor, LSF, PBS Pro, or Symphony), on Azure

CycleCloud allows you to:

- Deploy full clusters and other resources, including scheduler, compute VMs, storage, networking, and cache
- Orchestrate job, data, and cloud workflows
- Give admins full control over which users can run jobs, as well as where and at what cost
- Customize and optimize clusters through advanced policy and governance features, including cost controls, Active Directory integration, monitoring, and reporting
- Use your current job scheduler and applications without modification
- Take advantage of built-in autoscaling and battle-tested reference architectures for a wide range of HPC workloads and industries

Workload managers

The following are examples of cluster and workload managers that can run in Azure infrastructure. Create stand-alone clusters in Azure VMs or burst to Azure VMs from an on-premises cluster.

- Alces Flight Compute
- TIBCO DataSynapse GridServer
- Bright Cluster Manager
- IBM Spectrum Symphony and Symphony LSF
- Altair PBS Works
- Rescale
- Univa Grid Engine
- Microsoft HPC Pack
  - HPC Pack for Windows
  - HPC Pack for Linux

Containers

Containers can also be used to manage some HPC workloads. Services like the Azure Kubernetes Service (AKS) makes it simple to deploy a managed Kubernetes cluster in Azure.

- Azure Kubernetes Service (AKS)
- Container Registry

Cost management

Managing your HPC cost on Azure can be done through a few different ways. Ensure you've reviewed the Azure purchasing options to find the method that works best for your organization.

Security

For an overview of security best practices on Azure, review the Azure Security Documentation.

In addition to the network configurations available in the Cloud Bursting section, you may want to implement a hub/spoke configuration to isolate your compute resources:

- **Implement a hub-spoke network topology in Azure**

  The hub is a virtual network (VNet) in Azure that acts as a central point of connectivity to your on-premises network. The spokes are VNets that peer with the hub, and can be used to isolate workloads.

- **Implement a hub-spoke network topology with shared services in Azure**

  This reference architecture builds on the hub-spoke reference architecture to include shared services in the hub that can be consumed by all spokes.

HPC applications

Run custom or commercial HPC applications in Azure. Several examples in this section are benchmarked to scale efficiently with additional VMs or compute cores. Visit the Azure Marketplace for ready-to-deploy solutions.

> **NOTE**
> Check with the vendor of any commercial application for licensing or other restrictions for running in the cloud. Not all vendors offer pay-as-you-go licensing. You might need a licensing server in the cloud for your solution, or connect to an on-premises license server.

Engineering applications

- Altair RADIOSS
- ANSYS CFD
- MATLAB Distributed Computing Server
- StarCCM+

Graphics and rendering

- Autodesk Maya, 3ds Max, and Arnold on Azure Batch

AI and deep learning

- Microsoft Cognitive Toolkit
- Batch Shipyard recipes for deep learning

MPI providers

- Microsoft MPI

Remote visualization

Run GPU-powered virtual machines in Azure in the same region as the HPC output for the lowest latency, access, and to visualize remotely through Azure Virtual Desktop, Citrix, or VMware Horizon.

- GPU-optimized virtual machine sizes
- Configure GPU acceleration for Azure Virtual Desktop
-

Linux virtual desktops with Citrix

Build a VDI environment for Linux desktops using Citrix on Azure.

-

Windows desktops using Azure Virtual Desktop on Azure

Build a VDI environment for Windows desktops using Azure Virtual Desktop on Azure.

Performance benchmarks

- Compute benchmarks

Customer stories

There are a number of customers who have seen great success by using Azure for their HPC workloads. You can find a few of these customer case studies below:

- AXA Global P&C
- Axioma

- d3View
- EFS
- Hymans Robertson
- MetLife
- Microsoft Research
- Milliman
- Mitsubishi UFJ Securities International
- NeuroInitiative
- Schlumberger
- Towers Watson

Other important information

- Ensure your vCPU quota has been increased before attempting to run large-scale workloads.

Next steps

For the latest announcements, see:

- Microsoft HPC and Batch team blog
- Visit the Azure blog.

Microsoft Batch Examples

These tutorials will provide you with details on running applications on Microsoft Batch

- Get started developing with Batch
- Use Azure Batch code samples
- Use low-priority VMs with Batch
- Run containerized HPC workloads with Batch Shipyard
- Run parallel R workloads on Batch
- Run on-demand Spark jobs on Batch
- Use compute-intensive VMs in Batch pools

Choose an Azure compute service for your application

11/19/2021 • 7 minutes to read • Edit Online

Azure offers a number of ways to host your application code. The term *compute* refers to the hosting model for the computing resources that your application runs on. The following flowchart will help you to choose a compute service for your application.

If your application consists of multiple workloads, evaluate each workload separately. A complete solution may incorporate two or more compute services.

Choose a candidate service

Use the following flowchart to select a candidate compute service.

Definitions:

- **"Lift and shift"** is a strategy for migrating a workload to the cloud without redesigning the application or making code changes. Also called *rehosting*. For more information, see Azure migration and modernization

center.

- **Cloud optimized** is a strategy for migrating to the cloud by refactoring an application to take advantage of cloud-native features and capabilities.

The output from this flowchart is a **starting point** for consideration. Next, perform a more detailed evaluation of the service to see if it meets your needs.

This article includes several tables which may help you to make these tradeoff decisions. Based on this analysis, you may find that the initial candidate isn't suitable for your particular application or workload. In that case, expand your analysis to include other compute services.

> **NOTE**
> Learn more about reviewing your compute requirements for cloud adoption, in the Microsoft Cloud Adoption Framework for Azure.

Understand the basic features

If you're not familiar with the Azure service selected in the previous step, read the overview documentation to understand the basics of the service.

- App Service. A managed service for hosting web apps, mobile app back ends, RESTful APIs, or automated business processes.
- Azure Spring Cloud. A managed service designed and optimized for hosting Spring Boot apps.
- Azure Kubernetes Service (AKS). A managed Kubernetes service for running containerized applications.
- Batch. A managed service for running large-scale parallel and high-performance computing (HPC) applications
- Container Instances. The fastest and simplest way to run a container in Azure, without having to provision any virtual machines and without having to adopt a higher-level service.
- Functions. A managed FaaS service.
- Service Fabric. A distributed systems platform that can run in many environments, including Azure or on premises.
- Virtual machines. Deploy and manage VMs inside an Azure virtual network.

Understand the hosting models

Cloud services, including Azure services, generally fall into three categories: IaaS, PaaS, or FaaS. (There is also SaaS, software-as-a-service, which is out of scope for this article.) It's useful to understand the differences.

Infrastructure-as-a-Service (IaaS) lets you provision individual VMs along with the associated networking and storage components. Then you deploy whatever software and applications you want onto those VMs. This model is the closest to a traditional on-premises environment, except that Microsoft manages the infrastructure. You still manage the individual VMs.

Platform-as-a-Service (PaaS) provides a managed hosting environment, where you can deploy your application without needing to manage VMs or networking resources. Azure App Service is a PaaS service.

Functions-as-a-Service (FaaS) goes even further in removing the need to worry about the hosting environment. In a FaaS model, you simply deploy your code and the service automatically runs it. Azure Functions is a FaaS service.

> **NOTE**
> Azure Functions is an Azure serverless compute offering. You may read Choose the right integration and automation services in Azure to know how this service compares with other Azure serverless offerings, such as Logic Apps which provides serverless workflows.

There is a spectrum from IaaS to pure PaaS. For example, Azure VMs can autoscale by using virtual machine scale sets. This automatic scaling capability isn't strictly PaaS, but it's the type of management feature found in PaaS services.

In general, there is a tradeoff between control and ease of management. IaaS gives the most control, flexibility, and portability, but you have to provision, configure and manage the VMs and network components you create. FaaS services automatically manage nearly all aspects of running an application. PaaS services fall somewhere in between.

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Application composition	Agnostic	Applications, containers	Applications, microservices	Services, guest executables, containers	Functions	Containers	Containers	Scheduled jobs
Density	Agnostic	Multiple apps per instance via app service plans	Multiple apps per service instance	Multiple services per VM	Serverless [1]	Multiple containers per node	No dedicated instances	Multiple apps per VM
Minimum number of nodes	1 [2]	1	2	5 [3]	Serverless [1]	3 [3]	No dedicated nodes	1 [4]
State management	Stateless or Stateful	Stateless	Stateless	Stateless or stateful	Stateless	Stateless or Stateful	Stateless	Stateless
Web hosting	Agnostic	Built in	Built in	Agnostic	Not applicable	Agnostic	Agnostic	No
Can be deployed to dedicated VNet?	Supported	Supported[5]	Supported	Supported	Supported [5]	Supported	Supported	Supported
Hybrid connectivity	Supported	Supported [6]	Supported	Supported	Supported [7]	Supported	Not supported	Supported

Notes

1. If using Consumption plan. If using App Service plan, functions run on the VMs allocated for your App

Service plan. See Choose the correct service plan for Azure Functions.

2. Higher SLA with two or more instances.
3. Recommended for production environments.
4. Can scale down to zero after job completes.
5. Requires App Service Environment (ASE).
6. Use Azure App Service Hybrid Connections.
7. Requires App Service plan or Azure Functions Premium plan.

DevOps

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Local debugging	Agnostic	IIS Express, others [1]	Visual Studio Code, Intellij, Eclipse	Local node cluster	Visual Studio or Azure Functions CLI	Minikube, others	Local container runtime	Not supported
Programming model	Agnostic	Web and API applications, WebJobs for background tasks	Spring Boot, Steeltoe	Guest executable, Service model, Actor model, Containers	Functions with triggers	Agnostic	Agnostic	Command line application
Application update	No built-in support	Deployment slots	Rolling upgrade, Blue-green deployment	Rolling upgrade (per service)	Deployment slots	Rolling update	Not applicable	

Notes

1. Options include IIS Express for ASP.NET or node.js (iisnode); PHP web server; Azure Toolkit for IntelliJ, Azure Toolkit for Eclipse. App Service also supports remote debugging of deployed web app.
2. See Resource Manager providers, regions, API versions and schemas.

Scalability

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Autoscaling	Virtual machine scale sets	Built-in service	Built-in service	Virtual machine scale sets	Built-in service	Pod auto-scaling[1], cluster auto-scaling[2]	Not supported	N/A

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
Load balancer	Azure Load Balancer	Integrated	Integrated	Azure Load Balancer	Integrated	Azure Load Balancer or Application Gateway	No built-in support	Azure Load Balancer
Scale limit[3]	Platform image: 1000 nodes per scale set, Custom image: 600 nodes per scale set	30 instances, 100 with App Service Environment	500 app instances in Standard	100 nodes per scale set	200 instances per Function app	100 nodes per cluster (default limit)	20 container groups per subscription (default limit).	20 core limit (default limit).

Notes

1. See Autoscale pods.
2. See Automatically scale a cluster to meet application demands on Azure Kubernetes Service (AKS).
3. See Azure subscription and service limits, quotas, and constraints.

Availability

CRITERIA	VIRTUAL MACHINES	APP SERVICE	AZURE SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
SLA	SLA for Virtual Machines	SLA for App Service	SLA for Azure Spring Cloud	SLA for Service Fabric	SLA for Functions	SLA for AKS	SLA for Container Instances	SLA for Azure Batch
Multi region failover	Traffic manager	Traffic manager	Traffic manager, Multi-Region Cluster	Azure Front Door	Traffic manager	Not supported	Not Supported	

For guided learning on Service Guarantees, review Core Cloud Services - Azure architecture and service guarantees.

Security

Review and understand the available security controls and visibility for each service

- App Service
- App Spring Cloud
- Azure Kubernetes Service
- Batch

- Container Instances
- Functions
- Service Fabric
- Virtual machine - Windows
- Virtual machine - LINUX

Other criteria

CRITERIA	VIRTUAL MACHINES	APP SERVICE	APP SPRING CLOUD	SERVICE FABRIC	AZURE FUNCTIONS	AZURE KUBERNETES SERVICE	CONTAINER INSTANCES	AZURE BATCH
SSL	Configured in VM	Supported	Supported	Supported	Supported	Ingress controller	Use sidecar container	Supported
Cost	Windows, Linux	App Service pricing	Azure Spring Cloud pricing	Service Fabric pricing	Azure Functions pricing	AKS pricing	Container Instances pricing	Azure Batch pricing
Suitable architecture styles	N-Tier, Big compute (HPC)	Web-Queue-Worker, N-Tier	Spring Boot, Microservices	Microservices, Event-driven architecture	Microservices, Event-driven architecture	Microservices, Event-driven architecture	Microservices, task automation, batch jobs	Big compute (HPC)

The output from this flowchart is a **starting point** for consideration. Next, perform a more detailed evaluation of the service to see if it meets your needs.

Consider limits and cost

Perform a more detailed evaluation looking at the following aspects of the service:

- Service limits
- Cost
- SLA
- Regional availability
- Compute comparison tables

Next steps

- Core Cloud Services - Azure compute options. This Microsoft Learn module explores how compute services can solve common business needs.

Azure Kubernetes Service (AKS) architecture design

11/19/2021 • 4 minutes to read • Edit Online

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. Azure Kubernetes Service (AKS) makes it simple to deploy a managed Kubernetes cluster in Azure.

Organizations are at various points in their understanding, rationalizing, and adoption of Kubernetes on Azure. Your organization's journey will likely follow a similar path to many other technologies you've adopted; learning, aligning your organization around roles & responsibilities, and deploying production-ready workloads. From there, you'll iterate; growing your product as your customer and business demands change.

Introduction to Azure Kubernetes Service (AKS)

If you're new to Kubernetes or AKS, the best place to learn about the service is with Microsoft Learn. Microsoft Learn is a free, online training platform that provides interactive learning for Microsoft products and more. The **Introduction to Kubernetes on Azure** learning path will provide you with foundational knowledge that will take you through core concepts of containers, AKS cluster management, and workload deployment.

Introduction to Kubernetes on Azure

Path to production

You understand the benefits and trade-offs of Kubernetes, and have decided that AKS is the best Azure compute platform for your workload. Your organizational controls have been put into place; you're ready to learn how to deploy production-ready clusters for your workload.

Microsoft's AKS Baseline Cluster is the starting point to help you build production-ready AKS clusters.

Microsoft's AKS Baseline Cluster

We recommend you start from the baseline implementation and modify it to align to your workload's specific needs.

Suite of baseline implementations

We've provided a set of more baseline implementations to illustrate how you can adopt and configure components of AKS Baseline Cluster for various scenarios.

Microservices

When running microservices in the baseline cluster, you'll need to configure network policies, pod autoscaling, and set up distributed tracing for observability.

Microservices architecture using the baseline implementation

High security compliance

If you need a regulated environment, make the baseline implementation highly secure and restrict interactions to and from of the cluster. This use case is demonstrated in a cluster that's designed to run a PCI-DSS 3.2.1 workload.

Regulated baseline cluster for PCI-DSS 3.2.1

Business Continuity and Disaster Recovery

A resilient solution needs multiple instances of the baseline cluster across regions in an active/active and highly available configuration.

Baseline for multiregion clusters

Best practices

As organizations such as yours have adopted Azure, the Cloud Adoption Framework provides them prescriptive guidance as they move between the phases of the cloud adoption lifecycle. The Cloud Adoption Framework includes tools, programs, and content to simplify adoption of Kubernetes and related cloud-native practices at scale.

Kubernetes in the Cloud Adoption Framework

As part of on going operations, you may wish to spot check your cluster against current recommended best practices. The best place to start is to ensure your cluster is aligned with Microsoft's AKS Baseline Cluster.

See Best Practices for Cluster Operations and Best Practices for AKS Workloads.

> You may also consider evaluating a community-driven utility like The AKS Checklist as a way of organizing and tracking your alignment to these best practices.

Operations guide

Getting your workload deployed on AKS is a great milestone and this is when day-2 operations are going to be top-of-mind. **Microsoft's AKS Day 2 Operations Guide** was built for your ease of reference. This will help ensure you are ready to meet the demands of your customers and ensure you are prepared for break-fix situations via optimized triage processes.

Microsoft's AKS Day 2 Operations Guide

Stay current with AKS

Kubernetes and AKS are both moving fast. The platform is evolving and just knowing what's on the roadmap might help you make architectural decisions and understand planned deprecations; consider bookmarking it.

AKS product roadmap

Additional resources

The typical AKS solution journey shown ranges from learning about AKS to growing your existing clusters to meet new product and customer demands. However, you might also just be looking for additional reference and supporting material to help along the way for your specific situation.

Example solutions

If you're seeking additional references that use AKS as their foundation, here are a few to consider.

- Microservices architecture on AKS
- Secure DevOps for AKS
- Building a telehealth system
- CI/CD pipeline for container-based workloads

Azure Arc

Azure Kubernetes Service offers you a managed Kubernetes experience on Azure, however there are workloads

or situations that might be best suited for placing your own Kubernetes clusters under Azure Arc management. This includes your clusters such as RedHat OpenShift, RedHat RKE, and Canonical Charmed Kubernetes. Azure Arc management should also be used for AKS Engine clusters running in your datacenter, in another cloud, or on Azure Stack Hub.

Azure Arc enabled Kubernetes

Managed service provider

If you're a managed service provider, you already use Azure Lighthouse to manage resources for multiple customers. Azure Kubernetes Service supports Azure Lighthouse so that you can manage hosted Kubernetes environments and deploy containerized applications within your customers' tenants.

AKS with Azure Lighthouse

AWS or GCP professionals

These articles provide service mapping and comparison between Azure and other cloud services. This reference can help you ramp up quickly on Azure.

- Containers and container orchestrators for AWS Professionals
- Containers and container orchestrators for GCP Professionals

Azure Kubernetes Services (AKS) day-2 operations guide

11/19/2021 • 2 minutes to read • Edit Online

After you release an Azure Kubernetes Service (AKS)-hosted application, prepare for *day-2 operations*. Day-2 operations include triage, ongoing maintenance of deployed assets, rolling out upgrades, and troubleshooting.

Day-2 operations help you:

- Keep up to date with your service-level agreement (SLA) or service-level objective (SLO) requirements.
- Troubleshoot customer support requests.
- Stay current with the latest platform features and security updates.
- Plan for future growth.

Prerequisites

The Day-2 operations guide assumes that you've deployed the Azure Kubernetes Service (AKS) baseline architecture as an example of a production cluster.

Triage practices

11/19/2021 • 2 minutes to read • Edit Online

It's often challenging to do root-cause analysis given the different aspects of an AKS cluster. When triaging issues, consider a top-down approach on the cluster hierarchy. Start at the cluster level and drill down if necessary.

In the triage practices series, we'll walk you through the thought process of this approach. The articles show examples using a set of tools and dashboards, and how they can highlight some symptoms.

Common causes addressed in this series include:

- Network and connectivity problems caused by improper configuration.
- Control plane to node communication is broken.
- Kubelet pressures caused by insufficient compute, memory, or storage resources.
- DNS resolution issues.
- Node health is running out of disk IOPS.
- Admission control pipeline is blocking a large number of requests to the API server.
- The cluster doesn't have permissions to pull from the appropriate container registry.

This series isn't intended to resolve specific issues. For information about troubleshooting specific issues, see AKS Common Issues.

In the triage practices series

STEP	DESCRIPTION
1- Check the AKS cluster health	Start by checking the cluster the health of the overall cluster and networking.
2- Examine the node and pod health	Check the health of the AKS worker nodes.
3- Check the workload deployments	Check to see that all deployments and daemonSets are running.
4- Validate the admission controllers	Check whether the admission controllers are working as expected.
5- Verify the connection to the container registry	Verify the connection to the container registry.

Related links

- Day-2 operations
- AKS periscope
- AKS roadmap
- AKS documentation

Check the AKS cluster health

11/19/2021 • 2 minutes to read • Edit Online

Start by checking the health of the overall cluster and networking.

This article is part of a series. Read the introduction here.

Tools:

AKS Diagnostics. In Azure portal, navigate to the AKS cluster resource. Select **Diagnose and solve problems**.

Diagnostics shows a list of results from various test runs. If there are any issues found, **More info** can show you information about the underlying issue.

This image indicates that network and connectivity issues are caused by Azure CNI subnet configuration.

To learn more about this feature, see Azure Kubernetes Service Diagnostics overview.

Next steps

Examine the node and pod health

Examine the node and pod health

11/19/2021 • 5 minutes to read • Edit Online

If the cluster checks are clear, check the health of the AKS worker nodes. Determine the reason for the unhealthy node and resolve the issue.

This article is part of a series. Read the introduction here.

1- Check the overall health of the worker nodes

A node can be unhealthy because of various reasons. A common reason is when the control plane to node communication is broken as a result of misconfiguration of routing and firewall rules. As a fix, you can allow the necessary ports and fully qualified domain names through the firewall according to the AKS egress traffic guidance. Another reason can be kubelet pressures. In that case, add more compute, memory, or storage resources.

Tools:

You can check node health in one of these ways:

- **Azure Monitor - Containers health view**. In Azure portal, open Azure Monitor. Select **Containers**. On the right pane, select **Monitored clusters**. Select the cluster to view the health of the nodes, user pods, and system pods.

- **AKS - Nodes view:** In Azure portal, open navigate to the cluster. Select **Insights** under **Monitoring**. View **Nodes** on the right pane.

- **Prometheus and Grafana Dashboard**. Open the **Node Conditions** dashboard.

2- Verify the control plane and worker node connectivity

If worker nodes are healthy, examine the connectivity between the managed AKS control plane and the worker nodes. Depending on the age and type of cluster configuration, the connectivity pods are either **tunnelfront** or **aks-link**, and located in the **kube-system** namespace.

Tools:

- `kubectl`
- **Azure Monitor for Containers**

```
(⎈ |akskh20200929-admin:kube-system): repos  →  kubectl get pods | grep aks-link
aks-link-9947495dd-cqp2v                          2/2     Running   0          26m
```

If **tunnelfront** or **aks-link** connectivity is not working, establish connectivity after checking that the appropriate AKS egress traffic rules have been allowed. Here are the steps:

1. Restart **tunnelfront** or **aks-link**.

   ```
   kubectl rollout restart deploy aks-link
   ```

   If restarting the pods doesn't fix the connection, continue to the next step.

2. Check the logs and look for abnormalities. This output shows logs for a working connection.

```
kubectl logs -l app=aks-link -c openvpn-client --tail=50
```

```
(* |akskh20200929-admin:kube-system): repos  →  kubectl logs aks-link-9947495dd-cqp2v openvpn-client
Tue Sep 29 13:16:29 2020 WARNING: file '/etc/openvpn/certs/client.key' is group or others accessible
Tue Sep 29 13:16:29 2020 OpenVPN 2.4.4 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO] [LZ4] [EPOLL] [PKCS11] [MH/PKTINFO] [AEAD] built on May 14 2019
Tue Sep 29 13:16:29 2020 library versions: OpenSSL 1.1.1  11 Sep 2018, LZO 2.08
Tue Sep 29 13:16:29 2020 TCP/UDP: Preserving recently used remote address: [AF_INET]52.151.227.62:1194
Tue Sep 29 13:16:29 2020 UDP link local: (not bound)
Tue Sep 29 13:16:29 2020 UDP link remote: [AF_INET]52.151.227.62:1194
Tue Sep 29 13:16:29 2020 NOTE: UID/GID downgrade will be delayed because of --client, --pull, or --up-delay
Tue Sep 29 13:16:31 2020 [openvpn-server.5f7319c92f44fb00017807e0] Peer Connection Initiated with [AF_INET]52.151.227.62:1194
Tue Sep 29 13:16:32 2020 TUN/TAP device tun0 opened
Tue Sep 29 13:16:32 2020 do_ifconfig, tt->did_ifconfig_ipv6_setup=0
Tue Sep 29 13:16:32 2020 /sbin/ip link set dev tun0 up mtu 1500
Tue Sep 29 13:16:32 2020 /sbin/ip addr add dev tun0 192.0.2.2/24 broadcast 192.0.2.255
Tue Sep 29 13:16:32 2020 GID set to nogroup
Tue Sep 29 13:16:32 2020 UID set to nobody
Tue Sep 29 13:16:32 2020 WARNING: this configuration may cache passwords in memory -- use the auth-nocache option to prevent this
Tue Sep 29 13:16:32 2020 Initialization Sequence Completed
```

You can also retrieve those logs by searching the container logs in the logging and monitoring service. This example searches Azure Monitor for Containers to check for **aks-link** connectivity errors.

```
let ContainerIDs = KubePodInventory
| where ClusterId =~
"/subscriptions/YOUR_SUBSCRIPTION_ID/resourceGroups/RESOURCE_GROUP/providers/Microsoft.ContainerService/mana
gedClusters/YOUR_CLUSTER_ID"
| where Name has "aks-link"
| distinct ContainerID;
ContainerLog
| where ContainerID in (ContainerIDs)
| project LogEntrySource, LogEntry, TimeGenerated, Computer, Image, Name, ContainerID
| order by TimeGenerated desc
| limit 200
```

Here's another example query to check for **tunnelfront** connectivity errors.

```
let ContainerIDs = KubePodInventory
| where ClusterId =~
"/subscriptions/YOUR_SUBSCRIPTION_ID/resourceGroups/RESOURCE_GROUP/providers/Microsoft.ContainerService/mana
gedClusters/YOUR_CLUSTER_ID"
| where Name has "tunnelfront"
| distinct ContainerID;
ContainerLog
| where ContainerID in (ContainerIDs)
| where LogEntry has "ssh to tunnelend is not connected"
| project LogEntrySource, LogEntry, TimeGenerated, Computer, Image, Name, ContainerID
| order by TimeGenerated desc
| limit 200
```

If you can't get the logs through the kubectl or queries, use SSH into the node. This example finds the **tunnelfront** pod after connecting to the node through SSH.

```
kubectl pods -n kube-system -o wide | grep tunnelfront
ssh azureuser@<agent node pod is on, output from step above>
docker ps | grep tunnelfront
docker logs …
nslookup <ssh-server_fqdn>
ssh -vv azureuser@<ssh-server_fqdn> -p 9000
docker exec -it <tunnelfront_container_id> /bin/bash -c "ping bing.com"
kubectl get pods -n kube-system -o wide | grep <agent_node_where_tunnelfront_is_running>
kubectl delete po <kube_proxy_pod> -n kube-system
```

3- Validate DNS resolution when restricting egress

DNS resolution is a critical component of your cluster. If DNS resolution isn't working, then control plane errors

or container image pull failures may occur.

Tools:

- `nslookup`
- `dig`

Follow these steps to make sure that DNS resolution is working.

1. Exec into the pod to examine and use `nslookup` or `dig` if those tools are installed on the pod.
2. If the pod doesn't have those tools, start a utility pod in the same namespace and retry with the tools.
3. If those steps don't show insights, SSH to one of the nodes and try resolution from there. This step will help determine if the issue is related to AKS related or networking configuration.
4. If DNS resolves from the node, then the issue is related to Kubernetes DNS and not a networking issue. Restart Kubernetes DNS and check whether the issue is resolved. If not, open a Microsoft support ticket.
5. If DNS doesn't resolve from the node, then check the networking setup again to make sure that the appropriate routing paths and ports are open.

4- Check for kubelet errors

Check the kubelet process running on each worker node and make sure it's not experiencing any pressures. Those pressures can be related to CPU, memory, or storage.

Tools:

- **AKS - Kubelet Workbook**

- **Prometheus and Grafana Dashboard:** Kubelet Dashboard

The pressure increases when kubelet restarts and causes some sporadic, unpredictable behavior. Make sure that the error count isn't continuously growing. An occasional error is acceptable but a constant growth indicates an underlying issue that needs to be investigated and resolved.

5- Check disk IOPS for throttling

Check to see that file operations (IOPS) are not getting throttled and impacting services in the cluster.

Tools:

- **Azure Monitor for Containers Disk IO Workbook**

- **Prometheus and Grafana Dashboard:** Node Disk Dashboard

Physical storage devices have limitations, bandwidth, and total number of file operations. Azure Disks are used to store the OS running on the AKS nodes. They are subject to the same physical storage limitations.

Another way is to look at the throughput. IOPSs is measured with the average IO size * IOPS as the throughput in MB/s. Large IO sizes will lead to lower IOPS because the throughput of a disk doesn't change.

When a workload exceeds Azure Disks' service limits on Max IOPS, the cluster becomes unresponsive and blocked in IO Wait. Everything on Linux is a file. This includes network sockets, CNI, Docker, and other services that use network I/O. All of those files will fail if they're unable to read the disk.

The events that can trigger IOPS throttle include:

- High volume of containers running on the nodes. Docker IO is shared on the OS disk.
- Custom or third-party tools used for security, monitoring, logging that are running on the OS disk.
- Node failover events, and periodic jobs. As the load increases or the pods are scaled, this throttling occurs more frequently until all nodes go to **NotReady** state while the IO completes.

Related links

Virtual machine disk limits

Relationship between Virtual Machine and Disk Performance

Next steps

Check the workload deployments

Check the workload deployments

11/19/2021 • 2 minutes to read • Edit Online

Check to see that all deployments and daemonSets are running. The **Ready** and the **Available** matches the expected count.

This article is part of a series. Read the introduction here.

Tools:

- **AKS - Workloads**. In Azure portal, navigate to the AKS cluster resource. Select **Workloads**.

- **Prometheus and Grafana Dashboard**. Deployment Status Dashboard. This image is from Grafana Community Chart 10856.

Next steps

Validate the admission controllers

Validate the admission controllers are working as expected

11/19/2021 • 2 minutes to read • Edit Online

Check whether the admission controllers are working as expected.

This article is part of a series. Read the introduction here.

Issues because of admission controllers are an edge case but they should be considered. Here are some examples:

- Mutating and validating webhooks. Be careful when you add mutating and validating webhooks in your cluster. Make sure that they're highly available so that an unhealthy node doesn't cause API server requests to be blocked. AKS Policy, also known as OPA Gatekeeper, is an example of this type of webhooks. If there are problems in the admission control pipeline, it can block a large number of requests to the API server.
- Service meshes. They use admission controllers to automatically inject sidecars for example.

Tools `kubectl`

These commands check if AKS Policy is running in your cluster and how to validate that all of the admission controllers are functioning as expected.

```
# Check AKS Policy pods are running.
kubectl get po -n gatekeeper-system

# Sample Output
...
NAME                                      READY   STATUS    RESTARTS   AGE
gatekeeper-audit-65844778cb-rkflg         1/1     Running   0          163m
gatekeeper-controller-78797d4687-4pf6w    1/1     Running   0          163m
gatekeeper-controller-78797d4687-splzh    1/1     Running   0          163m
...
```

If this command doesn't run as expected, it could indicate that an admission controller, API service, or CRD isn't functioning correctly.

```
# Check that all API Resources are working correctly.
kubectl api-resources

# Sample Output
...
NAME                                      SHORTNAMES    APIGROUP                       NAMESPACED   KIND
bindings                                                                             true         Binding
componentstatuses                         cs                                         false
ComponentStatus
configmaps                                cm                                         true         ConfigMap
...
```

Next steps

Verify the connection to the container registry

Verify the connection to the container registry

11/19/2021 • 2 minutes to read • Edit Online

Make sure that the worker nodes have the correct permission to pull the necessary container images from the container registry.

This article is part of a series. Read the introduction here.

A common symptom of this issue is receiving **ImagePullBackoff** errors when getting or describing a pod. Be sure that the registry and image name are correct. Also, the cluster has permissions to pull from the appropriate container registry.

If you are using Azure Container Registry (ACR), the cluster service principal or managed identity should be granted **AcrPull** permissions against the registry.

One way is to run this command using the managed identity of the AKS cluster node pool. This command gets a list of its permissions.

```
# Get Kubelet Identity (Nodepool MSI)
ASSIGNEE=$(az aks show -g $RESOURCE_GROUP -n $NAME --query identityProfile.kubeletidentity.clientId -o tsv)
az role assignment list --assignee $ASSIGNEE --all -o table

# Expected Output
...
e5615a90-1767-4a4f-83b6-cecfa0675970  AcrPull
/subscriptions/.../providers/Microsoft.ContainerRegistry/registries/akskhacr
...
```

If you're using another container registry, check the appropriate **ImagePullSecret** credentials for the registry.

Related links

Import container images to a container registry

AKS Roadmap

Patching and upgrade guidance

11/19/2021 • 5 minutes to read • Edit Online

This section of the Azure Kubernetes Service (AKS) day-2 operations guide describes patching and upgrading practices for AKS worker nodes and Kubernetes (K8S) versions.

Node image upgrades

Microsoft provides patches and new images for image nodes weekly, but doesn't automatically patch them by default. Because AKS isn't a platform-as-a-service (PaaS), components like agent nodes have shared responsibility, and users must help maintain the AKS cluster. For example, applying an agent node operating system (OS) security patch requires user input.

AKS supports upgrading node images by using az aks nodepool upgrade, so you can keep up with the newest OS and runtime updates. To keep your agent node OS and runtime components patched, consider checking and applying node image upgrades bi-weekly, or automating the node image upgrade process. For more information about automating node image upgrades, see Node upgrade GitHub Actions.

An updated node image contains up-to-date OS security patches, kernel updates, Kubernetes security updates, newer versions of binaries like `kubelet`, and component version updates listed in the release notes. Node image updates have all relevant and validated security updates and feature updates. Using the node image upgrade method ensures you get only tested kernels and components that are compatible with those kernels.

You can use node image upgrades to streamline upgrades for both Windows and Linux node pools, but the processes differ slightly. Linux might receive daily security updates, but Windows Server nodes update by performing an AKS upgrade that deploys new nodes with the latest base Window Server image and patches.

Use kubectl describe nodes to check the OS kernel version and the OS image version of the nodes in your cluster:

```
kubectl describe nodes <NodeName>
```

Example output:

```
System Info:
  Machine ID:                 12345678-1234-1234-1234-0123456789ab
  System UUID:                abcdefga-abcd-abcd-abcd-abcdefg01234
  Boot ID:                    abcd0123-ab01-01ab-ab01-abcd01234567
  Kernel Version:             4.15.0-1096-azure
  OS Image:                   Ubuntu 16.04.7 LTS
  Operating System:           linux
  Architecture:               amd64
  Container Runtime Version:  docker://19.3.12
  Kubelet Version:            v1.17.9
  Kube-Proxy Version:         v1.17.9
```

Use the Azure CLI az aks nodepool list command to check the current node image versions of the nodes in a cluster:

```
az aks nodepool list \
   --resource-group <ResourceGroupName> --cluster-name <AKSClusterName> \
   --query "[].{Name:name,NodeImageVersion:nodeImageVersion}" --output table
```

Example output:

```
Name          NodeImageVersion
------------  ------------------------
systempool    AKSUbuntu-1604-2020.09.30
usernodepool  AKSUbuntu-1604-2020.09.30
usernp179     AKSUbuntu-1604-2020.10.28
```

Use az aks nodepool get-upgrades to find out the latest available node image version:

```
az aks nodepool get-upgrades \
   --resource-group <ResourceGroupName> --cluster-name <AKSClusterName> \
   --nodepool-name <NodePoolName> --output table
```

Example output:

```
KubernetesVersion  LatestNodeImageVersion     Name     OsType
-----------------  -------------------------  -------  ------
1.16.13            AKSUbuntu-1604-2020.11.11  default  Linux
```

To upgrade node pools to the latest node image version:

- Upgrade all nodes in node pools.
- Upgrade a specific node pool.
- Automate node pool upgrades using GitHub Actions.
- Automate node pool upgrades using auto-upgrade channels.

Cluster upgrades

The Kubernetes community releases minor K8S versions roughly every three months. The AKS GitHub release notes page publishes information about new AKS versions and releases, the latest AKS features, behavioral changes, bug fixes, and component updates. You can also subscribe to the GitHub AKS RSS feed.

The window of supported K8S versions on AKS is called "N - 2": (N (latest release) - 2 (minor versions)). It's important to establish a continuous cluster upgrade process to ensure that your AKS clusters don't go out of support. Once a new version becomes available, ideally you should plan an upgrade across all environments before the version becomes the default. This approach provides more control and predictability, and lets you plan upgrades with minimal disruption to existing workloads.

To minimize disruption to existing workloads during an upgrade:

- Set up multiple environments.
- Plan and schedule maintenance windows.
- Plan your tolerance for disruption.
- Use surge upgrades to control node pool upgrades.

To check when your cluster requires an upgrade, use az aks get-upgrades to get a list of available target upgrade versions for your AKS control plane. Determine the target version for your control plane from the results.

```
az aks get-upgrades \
   --resource-group <ResourceGroupName> --name <AKSClusterName> --output table
```

Example output:

```
MasterVersion  Upgrades
-------------  --------------------------------
1.17.9         1.17.11, 1.17.13, 1.18.8, 1.18.10
```

Check the Kubernetes versions of the nodes in your node pools to determine the node pools that need to be upgraded.

```
az aks nodepool list \
   --resource-group <ResourceGroupName> --cluster-name <AKSClusterName> \
   --query "[].{Name:name,k8version:orchestratorVersion}" --output table
```

Example output:

```
Name          K8version
------------  ------------
systempool    1.16.13
usernodepool  1.16.13
usernp179     1.17.9
```

You can upgrade the control plane first, and then upgrade the individual node pools.

1. Run the az aks upgrade command with the `--control-plane-only` flag to upgrade only the cluster control plane, and not any of the associated node pools:

   ```
   az aks upgrade \
      --resource-group <ResourceGroupName> --name <AKSClusterName> \
      --control-plane-only --no-wait \
      --kubernetes-version <KubernetesVersion>
   ```

2. Run az aks nodepool upgrade to upgrade node pools to the target version:

   ```
   az aks nodepool upgrade \
      --resource-group <ResourceGroupName> --cluster-name <AKSClusterName> --name <NodePoolName> \
      --no-wait --kubernetes-version <KubernetesVersion>
   ```

For information about validation rules for cluster upgrades, see Validation rules for upgrades.

Considerations

The following table describes characteristics of various AKS upgrade and patching scenarios:

SCENARIO	USER INITIATED	K8S UPGRADE	OS KERNEL UPGRADE	NODE IMAGE UPGRADE
Security patching	No	No	Yes, following reboot	Yes

SCENARIO	USER INITIATED	K8S UPGRADE	OS KERNEL UPGRADE	NODE IMAGE UPGRADE
Cluster create	Yes	Maybe	Yes, if an updated node image uses an updated kernel.	Yes, relative to an existing cluster if a new release is available.
Control plane K8S upgrade	Yes	Yes	No	No
Node pool K8S upgrade	Yes	Yes	Yes, if an updated node image uses an updated kernel.	Yes, if a new release is available.
Node pool scale up	Yes	No	No	No
Node image upgrade	Yes	No	Yes, if an updated node image uses an updated kernel.	Yes
Cluster auto upgrade	No	Yes	No	No

- For more information about Linux Automatic Security Updates, see AutomaticSecurityUpdates.
- It's possible that an OS security patch applied as part of a node image upgrade will install a later version of the kernel than creating a new cluster.
- You can use the Agent Pools - Get Upgrade Profile API to determine the latest node image version.
- Node pool scale up uses the model associated with the virtual machine scale set at creation. OS kernels upgrade when security patches are applied and the nodes reboot.
- Cluster auto upgrade is in preview. For more information, see Set auto-upgrade channel.
- Node image auto upgrade is in development. For more information, see Automatic Node Image Upgrade for node versions.

See also

- AKS day-2 operations guide
- AKS triage practices
- AKS common issues

Related links

- AKS product documentation
- AKS Roadmap
- Defining Day-2 Operations

Choose a Kubernetes at the edge compute option

11/19/2021 • 6 minutes to read • Edit Online

This document discusses the trade-offs for various options available for extending compute on the edge. The following considerations for each Kubernetes option are covered:

- **Operational cost.** The expected labor required to maintain and operate the Kubernetes clusters.
- **Ease of configuration.** The level of difficulty to configure and deploy a Kubernetes cluster.
- **Flexibility.** A measure of how adaptable the Kubernetes option is to integrate a customized configuration with existing infrastructure at the edge.
- **Mixed node.** Ability to run a Kubernetes cluster with both Linux and Windows nodes.

Assumptions

- You are a cluster operator looking to understand different options for running Kubernetes at the edge and managing clusters in Azure.
- You have a good understanding of existing infrastructure and any other infrastructure requirements, including storage and networking requirements.

After reading this document, you'll be in a better position to identify which option best fits your scenario and the environment required.

Kubernetes choices at a glance

	OPERATIONAL COST	EASE OF CONFIGURATION	FLEXIBILITY	MIXED NODE	SUMMARY
Bare-metal Kubernetes	High**	Difficult**	High**	Yes	A ground-up configuration on any available infrastructure at location with the option to use Azure Arc for added Azure capabilities.
K8s on Azure Stack Edge Pro	Low	Easy	Low	Linux only	Kubernetes deployed on Azure Stack Edge appliance deployed at location.
AKS on HCI	Low	Easy	Medium	Yes	AKS deployed on Azure Stack HCI or Windows Server 2019.

*Other managed edge platforms (OpenShift, Tanzu, and so on) aren't in scope for this document.

**These values are based on using *kubeadm*, for the sake of simplicity. Different options for running bare-metal Kubernetes at the edge would alter the rating in these categories.

Bare-metal Kubernetes

Ground-up configuration of Kubernetes using tools like kubeadm on any underlying infrastructure.

The biggest constraints for bare-metal Kubernetes are around the specific needs and requirements of the organization. The opportunity to use any distribution, networking interface, and plugin means higher complexity and operational cost. But this offers the most flexible option for customizing your cluster.

Scenario

Often, *edge* locations have specific requirements for running Kubernetes clusters that aren't met with the other Azure solutions described in this document. Meaning this option is typically best for those unable to use managed services due to unsupported existing infrastructure, or those who seek to have maximum control of their clusters.

- This option can be especially difficult for those who are new to Kubernetes. This isn't uncommon for organizations looking to run edge clusters. Options like MicroK8s or k3s aim to flatten that learning curve.
- It's important to understand any underlying infrastructure and any integration that is expected to take place up front. This will help to narrow down viable options and to identify any gaps with the open-source tooling and/or plugins.
- Enabling clusters with Azure Arc presents a simple way to manage your cluster from Azure alongside other resources. This also brings other Azure capabilities to your cluster, including Azure Policy, Azure Monitor, Microsoft Defender for Cloud, and other services.
- Because cluster configuration isn't trivial, it's especially important to be mindful of CI/CD. Tracking and acting on upstream changes of various plugins, and making sure those changes don't affect the health of your cluster, becomes a direct responsibility. It's important for you to have a strong CI/CD solution, strong testing, and monitoring in place.

Tooling options

Cluster bootstrap:

- kubeadm: Kubernetes tool for creating ground-up Kubernetes clusters. Good for standard compute resources (Linux/Windows).
- MicroK8s: Simplified administration and configuration ("*LowOps*"), conformant Kubernetes by Canonical.
- k3s: Certified Kubernetes distribution built for Internet of Things (IoT) and edge computing.

Storage:

- Explore available CSI drivers: Many options are available to fit your requirements from cloud to local file shares.

Networking:

- A full list of available add-ons can be found here: Networking add-ons. Some popular options include Flannel, a simple overlay network, and Calico, which provides a full networking stack.

Considerations

Operational cost:

- Without the support that comes with managed services, it's up to the organization to maintain and operate

the cluster as a whole (storage, networking, upgrades, observability, application management). The operational cost is considered high.

Ease of configuration:

- Evaluating the many open-source options at every stage of configuration whether its networking, storage, or monitoring options is inevitable and can become complex. Requires more consideration for configuring a CI/CD for cluster configuration. Because of these concerns, the ease of configuration is considered difficult.

Flexibility:

- With the ability to use any open-source tool or plugin without any provider restrictions, bare-metal Kubernetes is highly flexible.

Kubernetes on Azure Stack Edge

Kubernetes cluster (a master VM and a worker VM) configured and deployed for you on your Azure Stack Edge Pro device.

Azure Stack Edge Pro devices deliver Azure capabilities like compute, storage, networking, and hardware-accelerated machine learning (ML) to any edge location. Kubernetes clusters can be created once the compute role is enabled on any of the Pro-GPU, Pro-R, and Mini-R devices. Managing upgrades of the Kubernetes cluster can be done using standard updates available for the device.

Scenario

Ideal for those with existing (Linux) IoT workloads or upgrading their compute for ML at the edge. This is a good option when it isn't necessary to have more granular control over the clusters.

- Admin permissions aren't granted by default. Although you can work with the product group to make certain exceptions, this makes it difficult to have finer control of your cluster.
- There is an extra cost if there isn't already an Azure Stack Edge device. Explore Azure Stack Edge devices and see if any fit your compute requirements.
- Calico, MetalLB, and CoreDNS are installed for Kubernetes networking on the device.
- Only `Linux` workloads are supported at this time.
- In addition to Kubernetes, Azure Stack Edge also comes with the IoT runtime, which means that workloads may also be deployed to your Azure Stack Edge clusters via IoT Edge.
- Support for two node clusters isn't currently available. This effectively means that this option is *not* a highly available (HA) solution.

Considerations

Operational cost:

- With the support that comes with the device, operational cost is minimal and is scoped to workload management.

Ease of configuration:

- Pre-configured and well-documented Kubernetes cluster deployment simplifies the configuration required compared to bare-metal Kubernetes.

Flexibility:

- Configuration is already set, and Admin permissions aren't granted by default. Product group involvement may be required beyond basic configuration, and the underlying infrastructure must be an Azure Stack Edge

Pro device, making this a less flexible option.

AKS on HCI

Note: This option is currently in **preview**.

AKS-HCI is a set of predefined settings and configurations that is used to deploy one or more Kubernetes clusters (with Windows Admin Center or PowerShell modules) on a multi-node cluster running either Windows Server 2019 Datacenter or Azure Stack HCI 20H2.

Scenario

Ideal for those who want a simplified and streamlined way to get a Microsoft-supported cluster on compatible devices (Azure Stack HCI or Windows Server 2019 Datacenter). Operations and configuration complexity are reduced at the expense of the flexibility when compared to the bare-metal Kubernetes option.

Considerations

At the time of this writing, the preview comes with many limitations (permissions, networking limitations, large compute requirements, and documentation gaps). Purposes other than evaluation and development are discouraged that this time.

Operational cost:

- Microsoft-supported cluster minimizes operational costs.

Ease of configuration:

- Pre-configured and well-documented Kubernetes cluster deployment simplifies the configuration required compared to bare-metal Kubernetes.

Flexibility:

- Cluster configuration itself is set, but Admin permissions are granted. The underlying infrastructure must either be Azure Stack HCI or Windows Server 2019. This option is more flexible than Kubernetes on Azure Stack Edge and less flexible than bare-metal Kubernetes.

Next steps

For more information, see the following articles:

- What is Azure IoT Edge
- Kubernetes on your Azure Stack Edge Pro GPU device
- Use IoT Edge module to run a Kubernetes stateless application on your Azure Stack Edge Pro GPU device
- Deploy a Kubernetes stateless application via kubectl on your Azure Stack Edge Pro GPU device
- AI at the edge with Azure Stack Hub
- Building a CI/CD pipeline for microservices on Kubernetes
- Use Kubernetes dashboard to monitor your Azure Stack Edge Pro GPU device

Azure Data Architecture Guide

11/19/2021 • 2 minutes to read • Edit Online

This guide presents a structured approach for designing data-centric solutions on Microsoft Azure. It is based on proven practices derived from customer engagements.

> **NOTE**
>
> Learn more about adopting your systems for data governance, analytics, and data management, in Cloud adoption for data management.

Introduction

The cloud is changing the way applications are designed, including how data is processed and stored. Instead of a single general-purpose database that handles all of a solution's data, *polyglot persistence* solutions use multiple, specialized data stores, each optimized to provide specific capabilities. The perspective on data in the solution changes as a result. There are no longer multiple layers of business logic that read and write to a single data layer. Instead, solutions are designed around a *data pipeline* that describes how data flows through a solution, where it is processed, where it is stored, and how it is consumed by the next component in the pipeline.

How this guide is structured

This guide is structured around two general categories of data solution, *traditional RDBMS workloads* and *big data solutions*.

Traditional RDBMS workloads. These workloads include online transaction processing (OLTP) and online analytical processing (OLAP). Data in OLTP systems is typically relational data with a predefined schema and a set of constraints to maintain referential integrity. Often, data from multiple sources in the organization may be consolidated into a data warehouse, using an ETL process to move and transform the source data.

Big data solutions. A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems. The data may be processed in batch or in real time. Big data solutions typically involve a large amount of non-relational data, such as key-value data, JSON documents, or time series data. Often traditional RDBMS systems are not well-suited to store this type of data. The term *NoSQL* refers to a family of databases designed to hold non-relational data. The term isn't quite accurate, because many non-relational data stores support SQL compatible queries. The term *NoSQL* stands for "Not only SQL".

These two categories are not mutually exclusive, and there is overlap between them, but we feel that it's a useful way to frame the discussion. Within each category, the guide discusses **common scenarios**, including relevant Azure services and the appropriate architecture for the scenario. In addition, the guide compares **technology choices** for data solutions in Azure, including open source options. Within each category, we describe the key selection criteria and a capability matrix, to help you choose the right technology for your scenario.

This guide is not intended to teach you data science or database theory — you can find entire books on those subjects. Instead, the goal is to help you select the right data architecture or data pipeline for your scenario, and then select the Azure services and technologies that best fit your requirements. If you already have an architecture in mind, you can skip directly to the technology choices.

Extract, transform, and load (ETL)

11/19/2021 • 5 minutes to read • Edit Online

A common problem that organizations face is how to gather data from multiple sources, in multiple formats, and move it to one or more data stores. The destination may not be the same type of data store as the source, and often the format is different, or the data needs to be shaped or cleaned before loading it into its final destination.

Various tools, services, and processes have been developed over the years to help address these challenges. No matter the process used, there is a common need to coordinate the work and apply some level of data transformation within the data pipeline. The following sections highlight the common methods used to perform these tasks.

Extract, transform, and load (ETL) process

Extract, transform, and load (ETL) is a data pipeline used to collect data from various sources, transform the data according to business rules, and load it into a destination data store. The transformation work in ETL takes place in a specialized engine, and often involves using staging tables to temporarily hold data as it is being transformed and ultimately loaded to its destination.

The data transformation that takes place usually involves various operations, such as filtering, sorting, aggregating, joining data, cleaning data, deduplicating, and validating data.

Often, the three ETL phases are run in parallel to save time. For example, while data is being extracted, a transformation process could be working on data already received and prepare it for loading, and a loading process can begin working on the prepared data, rather than waiting for the entire extraction process to complete.

Relevant Azure service:

- Azure Data Factory v2

Other tools:

- SQL Server Integration Services (SSIS)

Extract, load, and transform (ELT)

Extract, load, and transform (ELT) differs from ETL solely in where the transformation takes place. In the ELT pipeline, the transformation occurs in the target data store. Instead of using a separate transformation engine, the processing capabilities of the target data store are used to transform data. This simplifies the architecture by removing the transformation engine from the pipeline. Another benefit to this approach is that scaling the target data store also scales the ELT pipeline performance. However, ELT only works well when the target system is powerful enough to transform the data efficiently.

Typical use cases for ELT fall within the big data realm. For example, you might start by extracting all of the source data to flat files in scalable storage such as Hadoop distributed file system (HDFS) or Azure Data Lake Store. Technologies such as Spark, Hive, or PolyBase can then be used to query the source data. The key point with ELT is that the data store used to perform the transformation is the same data store where the data is ultimately consumed. This data store reads directly from the scalable storage, instead of loading the data into its own proprietary storage. This approach skips the data copy step present in ETL, which can be a time consuming operation for large data sets.

In practice, the target data store is a data warehouse using either a Hadoop cluster (using Hive or Spark) or a Azure Synapse Analytics. In general, a schema is overlaid on the flat file data at query time and stored as a table, enabling the data to be queried like any other table in the data store. These are referred to as external tables because the data does not reside in storage managed by the data store itself, but on some external scalable storage.

The data store only manages the schema of the data and applies the schema on read. For example, a Hadoop cluster using Hive would describe a Hive table where the data source is effectively a path to a set of files in HDFS. In Azure Synapse, PolyBase can achieve the same result — creating a table against data stored externally to the database itself. Once the source data is loaded, the data present in the external tables can be processed using the capabilities of the data store. In big data scenarios, this means the data store must be capable of massively parallel processing (MPP), which breaks the data into smaller chunks and distributes processing of the chunks across multiple machines in parallel.

The final phase of the ELT pipeline is typically to transform the source data into a final format that is more efficient for the types of queries that need to be supported. For example, the data may be partitioned. Also, ELT might use optimized storage formats like Parquet, which stores row-oriented data in a columnar fashion and provides optimized indexing.

Relevant Azure service:

- Azure Synapse
- HDInsight with Hive

- Azure Data Factory v2
- Oozie on HDInsight

Other tools:

- SQL Server Integration Services (SSIS)

Data flow and control flow

In the context of data pipelines, the control flow ensures orderly processing of a set of tasks. To enforce the correct processing order of these tasks, precedence constraints are used. You can think of these constraints as connectors in a workflow diagram, as shown in the image below. Each task has an outcome, such as success, failure, or completion. Any subsequent task does not initiate processing until its predecessor has completed with one of these outcomes.

Control flows execute data flows as a task. In a data flow task, data is extracted from a source, transformed, or loaded into a data store. The output of one data flow task can be the input to the next data flow task, and data flows can run in parallel. Unlike control flows, you cannot add constraints between tasks in a data flow. You can, however, add a data viewer to observe the data as it is processed by each task.

In the diagram above, there are several tasks within the control flow, one of which is a data flow task. One of the tasks is nested within a container. Containers can be used to provide structure to tasks, providing a unit of work. One such example is for repeating elements within a collection, such as files in a folder or database statements.

Relevant Azure service:

- Azure Data Factory v2

Other tools:

- SQL Server Integration Services (SSIS)

Technology choices

- Online Transaction Processing (OLTP) data stores
- Online Analytical Processing (OLAP) data stores
- Data warehouses
- Pipeline orchestration

Next steps

The following reference architectures show end-to-end ELT pipelines on Azure:

- Enterprise BI in Azure with Azure Synapse
- Automated enterprise BI with Azure Synapse and Azure Data Factory

Online analytical processing (OLAP)

11/19/2021 • 8 minutes to read • Edit Online

Online analytical processing (OLAP) is a technology that organizes large business databases and supports complex analysis. It can be used to perform complex analytical queries without negatively affecting transactional systems.

The databases that a business uses to store all its transactions and records are called online transaction processing (OLTP) databases. These databases usually have records that are entered one at a time. Often they contain a great deal of information that is valuable to the organization. The databases that are used for OLTP, however, were not designed for analysis. Therefore, retrieving answers from these databases is costly in terms of time and effort. OLAP systems were designed to help extract this business intelligence information from the data in a highly performant way. This is because OLAP databases are optimized for heavy read, low write workloads.

Semantic modeling

A semantic data model is a conceptual model that describes the meaning of the data elements it contains. Organizations often have their own terms for things, sometimes with synonyms, or even different meanings for the same term. For example, an inventory database might track a piece of equipment with an asset ID and a serial number, but a sales database might refer to the serial number as the asset ID. There is no simple way to relate these values without a model that describes the relationship.

Semantic modeling provides a level of abstraction over the database schema, so that users don't need to know the underlying data structures. This makes it easier for end users to query data without performing aggregates and joins over the underlying schema. Also, usually columns are renamed to more user-friendly names, so that the context and meaning of the data are more obvious.

Semantic modeling is predominately used for read-heavy scenarios, such as analytics and business intelligence (OLAP), as opposed to more write-heavy transactional data processing (OLTP). This is mostly due to the nature of a typical semantic layer:

- Aggregation behaviors are set so that reporting tools display them properly.
- Business logic and calculations are defined.
- Time-oriented calculations are included.
- Data is often integrated from multiple sources.

Traditionally, the semantic layer is placed over a data warehouse for these reasons.

There are two primary types of semantic models:

- **Tabular**. Uses relational modeling constructs (model, tables, columns). Internally, metadata is inherited from OLAP modeling constructs (cubes, dimensions, measures). Code and script use OLAP metadata.
- **Multidimensional**. Uses traditional OLAP modeling constructs (cubes, dimensions, measures).

Relevant Azure service:

- Azure Analysis Services

Example use case

An organization has data stored in a large database. It wants to make this data available to business users and customers to create their own reports and do some analysis. One option is just to give those users direct access to the database. However, there are several drawbacks to doing this, including managing security and controlling access. Also, the design of the database, including the names of tables and columns, may be hard for a user to understand. Users would need to know which tables to query, how those tables should be joined, and other business logic that must be applied to get the correct results. Users would also need to know a query language like SQL even to get started. Typically this leads to multiple users reporting the same metrics but with different results.

Another option is to encapsulate all of the information that users need into a semantic model. The semantic model can be more easily queried by users with a reporting tool of their choice. The data provided by the semantic model is pulled from a data warehouse, ensuring that all users see a single version of the truth. The semantic model also provides friendly table and column names, relationships between tables, descriptions, calculations, and row-level security.

Typical traits of semantic modeling

Semantic modeling and analytical processing tends to have the following traits:

REQUIREMENT	DESCRIPTION
Schema	Schema on write, strongly enforced
Uses Transactions	No
Locking Strategy	None
Updateable	No (typically requires recomputing cube)
Appendable	No (typically requires recomputing cube)

REQUIREMENT	DESCRIPTION
Workload	Heavy reads, read-only
Indexing	Multidimensional indexing
Datum size	Small to medium sized
Model	Multidimensional
Data shape:	Cube or star/snowflake schema
Query flexibility	Highly flexible
Scale:	Large (10s-100s GBs)

When to use this solution

Consider OLAP in the following scenarios:

- You need to execute complex analytical and ad hoc queries rapidly, without negatively affecting your OLTP systems.
- You want to provide business users with a simple way to generate reports from your data
- You want to provide a number of aggregations that will allow users to get fast, consistent results.

OLAP is especially useful for applying aggregate calculations over large amounts of data. OLAP systems are optimized for read-heavy scenarios, such as analytics and business intelligence. OLAP allows users to segment multi-dimensional data into slices that can be viewed in two dimensions (such as a pivot table) or filter the data by specific values. This process is sometimes called "slicing and dicing" the data, and can be done regardless of whether the data is partitioned across several data sources. This helps users to find trends, spot patterns, and explore the data without having to know the details of traditional data analysis.

Semantic models can help business users abstract relationship complexities and make it easier to analyze data quickly.

Challenges

For all the benefits OLAP systems provide, they do produce a few challenges:

- Whereas data in OLTP systems is constantly updated through transactions flowing in from various sources, OLAP data stores are typically refreshed at a much slower intervals, depending on business needs. This means OLAP systems are better suited for strategic business decisions, rather than immediate responses to changes. Also, some level of data cleansing and orchestration needs to be planned to keep the OLAP data stores up-to-date.
- Unlike traditional, normalized, relational tables found in OLTP systems, OLAP data models tend to be multidimensional. This makes it difficult or impossible to directly map to entity-relationship or object-oriented models, where each attribute is mapped to one column. Instead, OLAP systems typically use a star or snowflake schema in place of traditional normalization.

OLAP in Azure

In Azure, data held in OLTP systems such as Azure SQL Database is copied into the OLAP system, such as Azure Analysis Services. Data exploration and visualization tools like Power BI, Excel, and third-party options connect

to Analysis Services servers and provide users with highly interactive and visually rich insights into the modeled data. The flow of data from OLTP data to OLAP is typically orchestrated using SQL Server Integration Services, which can be executed using Azure Data Factory.

In Azure, all of the following data stores will meet the core requirements for OLAP:

- SQL Server with Columnstore indexes
- Azure Analysis Services
- SQL Server Analysis Services (SSAS)

SQL Server Analysis Services (SSAS) offers OLAP and data mining functionality for business intelligence applications. You can either install SSAS on local servers, or host within a virtual machine in Azure. Azure Analysis Services is a fully managed service that provides the same major features as SSAS. Azure Analysis Services supports connecting to various data sources in the cloud and on-premises in your organization.

Clustered Columnstore indexes are available in SQL Server 2014 and above, as well as Azure SQL Database, and are ideal for OLAP workloads. However, beginning with SQL Server 2016 (including Azure SQL Database), you can take advantage of hybrid transactional/analytics processing (HTAP) through the use of updateable nonclustered columnstore indexes. HTAP enables you to perform OLTP and OLAP processing on the same platform, which removes the need to store multiple copies of your data, and eliminates the need for distinct OLTP and OLAP systems. For more information, see Get started with Columnstore for real-time operational analytics.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Do you require secure authentication using Azure Active Directory (Azure AD)?
- Do you want to conduct real-time analytics? If so, narrow your options to those that support real-time analytics.

  Real-time analytics in this context applies to a single data source, such as an enterprise resource planning (ERP) application, that will run both an operational and an analytics workload. If you need to integrate data from multiple sources, or require extreme analytics performance by using pre-aggregated data such as cubes, you might still require a separate data warehouse.
- Do you need to use pre-aggregated data, for example to provide semantic models that make analytics more business user friendly? If yes, choose an option that supports multidimensional cubes or tabular semantic models.

  Providing aggregates can help users consistently calculate data aggregates. Pre-aggregated data can also provide a large performance boost when dealing with several columns across many rows. Data can be pre-aggregated in multidimensional cubes or tabular semantic models.
- Do you need to integrate data from several sources, beyond your OLTP data store? If so, consider options that easily integrate multiple data sources.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE ANALYSIS SERVICES	SQL SERVER ANALYSIS SERVICES	SQL SERVER WITH COLUMNSTORE INDEXES	AZURE SQL DATABASE WITH COLUMNSTORE INDEXES
Is managed service	Yes	No	No	Yes
Supports multidimensional cubes	No	Yes	No	No
Supports tabular semantic models	Yes	Yes	No	No
Easily integrate multiple data sources	Yes	Yes	No [1]	No [1]
Supports real-time analytics	No	No	Yes	Yes
Requires process to copy data from source(s)	Yes	Yes	No	No
Azure AD integration	Yes	No	No [2]	Yes

[1] Although SQL Server and Azure SQL Database cannot be used to query from and integrate multiple external data sources, you can still build a pipeline that does this for you using SSIS or Azure Data Factory. SQL Server hosted in an Azure VM has additional options, such as linked servers and PolyBase. For more information, see Pipeline orchestration, control flow, and data movement.

[2] Connecting to SQL Server running on an Azure Virtual Machine is not supported using an Azure AD account. Use a domain Active Directory account instead.

Scalability Capabilities

CAPABILITY	AZURE ANALYSIS SERVICES	SQL SERVER ANALYSIS SERVICES	SQL SERVER WITH COLUMNSTORE INDEXES	AZURE SQL DATABASE WITH COLUMNSTORE INDEXES
Redundant regional servers for high availability	Yes	No	Yes	Yes
Supports query scale out	Yes	No	Yes	Yes
Dynamic scalability (scale up)	Yes	No	Yes	Yes

Online transaction processing (OLTP)

11/19/2021 • 6 minutes to read • Edit Online

The management of transactional data using computer systems is referred to as online transaction processing (OLTP). OLTP systems record business interactions as they occur in the day-to-day operation of the organization, and support querying of this data to make inferences.

Transactional data

Transactional data is information that tracks the interactions related to an organization's activities. These interactions are typically business transactions, such as payments received from customers, payments made to suppliers, products moving through inventory, orders taken, or services delivered. Transactional events, which represent the transactions themselves, typically contain a time dimension, some numerical values, and references to other data.

Transactions typically need to be *atomic* and *consistent*. Atomicity means that an entire transaction always succeeds or fails as one unit of work, and is never left in a half-completed state. If a transaction cannot be completed, the database system must roll back any steps that were already done as part of that transaction. In a traditional RDBMS, this rollback happens automatically if a transaction cannot be completed. Consistency means that transactions always leave the data in a valid state. (These are very informal descriptions of atomicity and consistency. There are more formal definitions of these properties, such as ACID.)

Transactional databases can support strong consistency for transactions using various locking strategies, such as pessimistic locking, to ensure that all data is strongly consistent within the context of the enterprise, for all users and processes.

The most common deployment architecture that uses transactional data is the data store tier in a 3-tier architecture. A 3-tier architecture typically consists of a presentation tier, business logic tier, and data store tier. A related deployment architecture is the N-tier architecture, which may have multiple middle-tiers handling business logic.

Typical traits of transactional data

Transactional data tends to have the following traits:

REQUIREMENT	DESCRIPTION
Normalization	Highly normalized
Schema	Schema on write, strongly enforced
Consistency	Strong consistency, ACID guarantees
Integrity	High integrity
Uses transactions	Yes
Locking strategy	Pessimistic or optimistic
Updateable	Yes

REQUIREMENT	DESCRIPTION
Appendable	Yes
Workload	Heavy writes, moderate reads
Indexing	Primary and secondary indexes
Datum size	Small to medium sized
Model	Relational
Data shape	Tabular
Query flexibility	Highly flexible
Scale	Small (MBs) to Large (a few TBs)

When to use this solution

Choose OLTP when you need to efficiently process and store business transactions and immediately make them available to client applications in a consistent way. Use this architecture when any tangible delay in processing would have a negative impact on the day-to-day operations of the business.

OLTP systems are designed to efficiently process and store transactions, as well as query transactional data. The goal of efficiently processing and storing individual transactions by an OLTP system is partly accomplished by data normalization — that is, breaking the data up into smaller chunks that are less redundant. This supports efficiency because it enables the OLTP system to process large numbers of transactions independently, and avoids extra processing needed to maintain data integrity in the presence of redundant data.

Challenges

Implementing and using an OLTP system can create a few challenges:

- OLTP systems are not always good for handling aggregates over large amounts of data, although there are exceptions, such as a well-planned SQL Server-based solution. Analytics against the data, that rely on aggregate calculations over millions of individual transactions, are very resource intensive for an OLTP system. They can be slow to execute and can cause a slow-down by blocking other transactions in the database.
- When conducting analytics and reporting on data that is highly normalized, the queries tend to be complex, because most queries need to de-normalize the data by using joins. Also, naming conventions for database objects in OLTP systems tend to be terse and succinct. The increased normalization coupled with terse naming conventions makes OLTP systems difficult for business users to query, without the help of a DBA or data developer.
- Storing the history of transactions indefinitely and storing too much data in any one table can lead to slow query performance, depending on the number of transactions stored. The common solution is to maintain a relevant window of time (such as the current fiscal year) in the OLTP system and offload historical data to other systems, such as a data mart or data warehouse.

OLTP in Azure

Applications such as websites hosted in App Service Web Apps, REST APIs running in App Service, or mobile or desktop applications communicate with the OLTP system, typically via a REST API intermediary.

In practice, most workloads are not purely OLTP. There tends to be an analytical component as well. In addition, there is an increasing demand for real-time reporting, such as running reports against the operational system. This is also referred to as HTAP (Hybrid Transactional and Analytical Processing). For more information, see Online Analytical Processing (OLAP).

In Azure, all of the following data stores will meet the core requirements for OLTP and the management of transaction data:

- Azure SQL Database
- SQL Server in an Azure virtual machine
- Azure Database for MySQL
- Azure Database for PostgreSQL

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Does your solution have specific dependencies for Microsoft SQL Server, MySQL or PostgreSQL compatibility? Your application may limit the data stores you can choose based on the drivers it supports for communicating with the data store, or the assumptions it makes about which database is used.
- Are your write throughput requirements particularly high? If yes, choose an option that provides in-memory tables.
- Is your solution multitenant? If so, consider options that support capacity pools, where multiple database instances draw from an elastic pool of resources, instead of fixed resources per database. This can help you better distribute capacity across all database instances, and can make your solution more cost effective.
- Does your data need to be readable with low latency in multiple regions? If yes, choose an option that supports readable secondary replicas.
- Does your database need to be highly available across geo-graphic regions? If yes, choose an option that supports geographic replication. Also consider the options that support automatic failover from the primary replica to a secondary replica.
- Does your database have specific security needs? If yes, examine the options that provide capabilities like row level security, data masking, and transparent data encryption.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Is Managed Service	Yes	No	Yes	Yes
Runs on Platform	N/A	Windows, Linux, Docker	N/A	N/A

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Programmability [1]	T-SQL, .NET, R	T-SQL, .NET, R, Python	SQL	SQL, PL/pgSQL

[1] Not including client driver support, which allows many programming languages to connect to and use the OLTP data store.

Scalability capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Maximum database instance size	4 TB	256 TB	16 TB	16 TB
Supports capacity pools	Yes	Yes	No	No
Supports clusters scale out	No	Yes	No	No
Dynamic scalability (scale up)	Yes	No	Yes	Yes

Analytic workload capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Temporal tables	Yes	Yes	No	No
In-memory (memory-optimized) tables	Yes	Yes	No	No
Columnstore support	Yes	Yes	No	No
Adaptive query processing	Yes	Yes	No	No

Availability capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Readable secondaries	Yes	Yes	Yes	Yes
Geographic replication	Yes	Yes	Yes	Yes
Automatic failover to secondary	Yes	No	No	No

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Point-in-time restore	Yes	Yes	Yes	Yes

Security capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN AN AZURE VIRTUAL MACHINE	AZURE DATABASE FOR MYSQL	AZURE DATABASE FOR POSTGRESQL
Row level security	Yes	Yes	Yes	Yes
Data masking	Yes	Yes	No	No
Transparent data encryption	Yes	Yes	Yes	Yes
Restrict access to specific IP addresses	Yes	Yes	Yes	Yes
Restrict access to allow VNet access only	Yes	Yes	Yes	Yes
Azure Active Directory authentication	Yes	No	Yes	Yes
Active Directory authentication	No	Yes	No	No
Multi-factor authentication	Yes	No	Yes	Yes
Supports Always Encrypted	Yes	Yes	No	No
Private IP	No	Yes	No	No

Data warehousing

11/19/2021 • 11 minutes to read • Edit Online

A data warehouse is a centralized repository of integrated data from one or more disparate sources. Data warehouses store current and historical data and are used for reporting and analysis of the data.

To move data into a data warehouse, data is periodically extracted from various sources that contain important business information. As the data is moved, it can be formatted, cleaned, validated, summarized, and reorganized. Alternatively, the data can be stored in the lowest level of detail, with aggregated views provided in the warehouse for reporting. In either case, the data warehouse becomes a permanent data store for reporting, analysis, and business intelligence (BI).

Data warehouse architectures

The following reference architectures show end-to-end data warehouse architectures on Azure:

- Enterprise BI in Azure with Azure Synapse Analytics. This reference architecture implements an extract, load, and transform (ELT) pipeline that moves data from an on-premises SQL Server database into Azure Synapse.
- Automated enterprise BI with Azure Synapse and Azure Data Factory. This reference architecture shows an ELT pipeline with incremental loading, automated using Azure Data Factory.

When to use this solution

Choose a data warehouse when you need to turn massive amounts of data from operational systems into a format that is easy to understand. Data warehouses don't need to follow the same terse data structure you may be using in your OLTP databases. You can use column names that make sense to business users and analysts, restructure the schema to simplify relationships, and consolidate several tables into one. These steps help guide users who need to create reports and analyze the data in BI systems, without the help of a database administrator (DBA) or data developer.

Consider using a data warehouse when you need to keep historical data separate from the source transaction systems for performance reasons. Data warehouses make it easy to access historical data from multiple locations, by providing a centralized location using common formats, keys, and data models.

Because data warehouses are optimized for read access, generating reports is faster than using the source transaction system for reporting.

Other benefits include:

- The data warehouse can store historical data from multiple sources, representing a single source of truth.
- You can improve data quality by cleaning up data as it is imported into the data warehouse.
- Reporting tools don't compete with the transactional systems for query processing cycles. A data warehouse allows the transactional system to focus on handling writes, while the data warehouse satisfies the majority of read requests.
- A data warehouse can consolidate data from different software.
- Data mining tools can find hidden patterns in the data using automatic methodologies.
- Data warehouses make it easier to provide secure access to authorized users, while restricting access to others. Business users don't need access to the source data, removing a potential attack vector.
- Data warehouses make it easier to create business intelligence solutions, such as OLAP cubes.

Challenges

Properly configuring a data warehouse to fit the needs of your business can bring some of the following challenges:

- Committing the time required to properly model your business concepts. Data warehouses are information driven. You must standardize business-related terms and common formats, such as currency and dates. You also need to restructure the schema in a way that makes sense to business users but still ensures accuracy of data aggregates and relationships.

- Planning and setting up your data orchestration. Consider how to copy data from the source transactional system to the data warehouse, and when to move historical data from operational data stores into the warehouse.

- Maintaining or improving data quality by cleaning the data as it is imported into the warehouse.

Data warehousing in Azure

You may have one or more sources of data, whether from customer transactions or business applications. This data is traditionally stored in one or more OLTP databases. The data could be persisted in other storage mediums such as network shares, Azure Storage Blobs, or a data lake. The data could also be stored by the data warehouse itself or in a relational database such as Azure SQL Database. The purpose of the analytical data store layer is to satisfy queries issued by analytics and reporting tools against the data warehouse. In Azure, this analytical store capability can be met with Azure Synapse, or with Azure HDInsight using Hive or Interactive Query. In addition, you will need some level of orchestration to move or copy data from data storage to the data warehouse, which can be done using Azure Data Factory or Oozie on Azure HDInsight.

There are several options for implementing a data warehouse in Azure, depending on your needs. The following lists are broken into two categories, symmetric multiprocessing (SMP) and massively parallel processing (MPP).

SMP:

- Azure SQL Database
- SQL Server in a virtual machine

MPP:

- Azure Synapse Analytics (formerly Azure Data Warehouse)
- Apache Hive on HDInsight
- Interactive Query (Hive LLAP) on HDInsight

As a general rule, SMP-based warehouses are best suited for small to medium data sets (up to 4-100 TB), while MPP is often used for big data. The delineation between small/medium and big data partly has to do with your organization's definition and supporting infrastructure. (See Choosing an OLTP data store.)

Beyond data sizes, the type of workload pattern is likely to be a greater determining factor. For example, complex queries may be too slow for an SMP solution, and require an MPP solution instead. MPP-based systems usually have a performance penalty with small data sizes, because of how jobs are distributed and consolidated across nodes. If your data sizes already exceed 1 TB and are expected to continually grow, consider selecting an MPP solution. However, if your data sizes are smaller, but your workloads are exceeding the available resources of your SMP solution, then MPP may be your best option as well.

The data accessed or stored by your data warehouse could come from a number of data sources, including a data lake, such as Azure Data Lake Storage. For a video session that compares the different strengths of MPP services that can use Azure Data Lake, see Azure Data Lake and Azure Data Warehouse: Applying Modern Practices to Your App.

SMP systems are characterized by a single instance of a relational database management system sharing all resources (CPU/Memory/Disk). You can scale up an SMP system. For SQL Server running on a VM, you can scale up the VM size. For Azure SQL Database, you can scale up by selecting a different service tier.

MPP systems can be scaled out by adding more compute nodes (which have their own CPU, memory, and I/O subsystems). There are physical limitations to scaling up a server, at which point scaling out is more desirable, depending on the workload. However, the differences in querying, modeling, and data partitioning mean that MPP solutions require a different skill set.

When deciding which SMP solution to use, see A closer look at Azure SQL Database and SQL Server on Azure VMs.

Azure Synapse (formerly Azure SQL Data Warehouse) can also be used for small and medium datasets, where the workload is compute and memory intensive. Read more about Azure Synapse patterns and common scenarios:

- Azure SQL Data Warehouse Workload Patterns and Anti-Patterns
- Azure SQL Data Warehouse loading patterns and strategies
- Migrating data to Azure SQL Data Warehouse in practice
- Common ISV application patterns using Azure SQL Data Warehouse

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Are you working with extremely large data sets or highly complex, long-running queries? If yes, consider an MPP option.
- For a large data set, is the data source structured or unstructured? Unstructured data may need to be processed in a big data environment such as Spark on HDInsight, Azure Databricks, Hive LLAP on HDInsight, or Azure Data Lake Analytics. All of these can serve as ELT (Extract, Load, Transform) and ETL (Extract, Transform, Load) engines. They can output the processed data into structured data, making it easier to load into Azure Synapse or one of the other options. For structured data, Azure Synapse has a performance tier called Optimized for Compute, for compute-intensive workloads requiring ultra-high performance.
- Do you want to separate your historical data from your current, operational data? If so, select one of the options where orchestration is required. These are standalone warehouses optimized for heavy read access, and are best suited as a separate historical data store.
- Do you need to integrate data from several sources, beyond your OLTP data store? If so, consider options

that easily integrate multiple data sources.

- Do you have a multitenancy requirement? If so, Azure Synapse is not ideal for this requirement. For more information, see Azure Synapse Patterns and Anti-Patterns.
- Do you prefer a relational data store? If so, choose an option with a relational data store, but also note that you can use a tool like PolyBase to query non-relational data stores if needed. If you decide to use PolyBase, however, run performance tests against your unstructured data sets for your workload.
- Do you have real-time reporting requirements? If you require rapid query response times on high volumes of singleton inserts, choose an option that supports real-time reporting.
- Do you need to support a large number of concurrent users and connections? The ability to support a number of concurrent users/connections depends on several factors.
  - For Azure SQL Database, refer to the documented resource limits based on your service tier.
  - SQL Server allows a maximum of 32,767 user connections. When running on a VM, performance will depend on the VM size and other factors.
  - Azure Synapse has limits on concurrent queries and concurrent connections. For more information, see Concurrency and workload management in Azure Synapse. Consider using complementary services, such as Azure Analysis Services, to overcome limits in Azure Synapse.
- What sort of workload do you have? In general, MPP-based warehouse solutions are best suited for analytical, batch-oriented workloads. If your workloads are transactional by nature, with many small read/write operations or multiple row-by-row operations, consider using one of the SMP options. One exception to this guideline is when using stream processing on an HDInsight cluster, such as Spark Streaming, and storing the data within a Hive table.

Capability Matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER (VM)	AZURE SYNAPSE	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Is managed service	Yes	No	Yes	Yes [1]	Yes [1]
Requires data orchestration (holds copy of data/historical data)	No	No	Yes	Yes	Yes
Easily integrate multiple data sources	No	No	Yes	Yes	Yes
Supports pausing compute	No	No	Yes	No [2]	No [2]
Relational data store	Yes	Yes	Yes	No	No

CAPABILITY	AZURE SQL DATABASE	SQL SERVER (VM)	AZURE SYNAPSE	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Real-time reporting	Yes	Yes	No	No	Yes
Flexible backup restore points	Yes	Yes	No [3]	Yes [4]	Yes [4]
SMP/MPP	SMP	SMP	MPP	MPP	MPP

[1] Manual configuration and scaling.

[2] HDInsight clusters can be deleted when not needed, and then re-created. Attach an external data store to your cluster so your data is retained when you delete your cluster. You can use Azure Data Factory to automate your cluster's lifecycle by creating an on-demand HDInsight cluster to process your workload, then delete it once the processing is complete.

[3] With Azure Synapse, you can restore a database to any available restore point within the last seven days. Snapshots start every four to eight hours and are available for seven days. When a snapshot is older than seven days, it expires and its restore point is no longer available.

[4] Consider using an external Hive metastore that can be backed up and restored as needed. Standard backup and restore options that apply to Blob Storage or Data Lake Storage can be used for the data, or third-party HDInsight backup and restore solutions, such as Imanis Data can be used for greater flexibility and ease of use.

Scalability capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER (VM)	AZURE SYNAPSE	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Redundant regional servers for high availability	Yes	Yes	Yes	No	No
Supports query scale out (distributed queries)	No	No	Yes	Yes	Yes
Dynamic scalability	Yes	No	Yes [1]	No	No
Supports in-memory caching of data	Yes	Yes	Yes	Yes	Yes

[1] Azure Synapse allows you to scale up or down by adjusting the number of data warehouse units (DWUs). See Manage compute power in Azure Synapse.

Security capabilities

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN A VIRTUAL MACHINE	AZURE SYNAPSE	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT

CAPABILITY	AZURE SQL DATABASE	SQL SERVER IN A VIRTUAL MACHINE	AZURE SYNAPSE	APACHE HIVE ON HDINSIGHT	HIVE LLAP ON HDINSIGHT
Authentication	SQL / Azure Active Directory (Azure AD)	SQL / Azure AD / Active Directory	SQL / Azure AD	local / Azure AD [1]	local / Azure AD [1]
Authorization	Yes	Yes	Yes	Yes	Yes [1]
Auditing	Yes	Yes	Yes	Yes	Yes [1]
Data encryption at rest	Yes [2]	Yes [2]	Yes [2]	Yes [2]	Yes [1]
Row-level security	Yes	Yes	Yes	No	Yes [1]
Supports firewalls	Yes	Yes	Yes	Yes	Yes [3]
Dynamic data masking	Yes	Yes	Yes	No	Yes [1]

[1] Requires using a domain-joined HDInsight cluster.

[2] Requires using Transparent Data Encryption (TDE) to encrypt and decrypt your data at rest.

[3] Supported when used within an Azure Virtual Network.

Read more about securing your data warehouse:

- Securing your SQL Database
- Secure a database in Azure Synapse
- Extend Azure HDInsight using an Azure Virtual Network
- Enterprise-level Hadoop security with domain-joined HDInsight clusters

Non-relational data and NoSQL

11/19/2021 • 12 minutes to read • Edit Online

A *non-relational database* is a database that does not use the tabular schema of rows and columns found in most traditional database systems. Instead, non-relational databases use a storage model that is optimized for the specific requirements of the type of data being stored. For example, data may be stored as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.

What all of these data stores have in common is that they don't use a relational model. Also, they tend to be more specific in the type of data they support and how data can be queried. For example, time series data stores are optimized for queries over time-based sequences of data. However, graph data stores are optimized for exploring weighted relationships between entities. Neither format would generalize well to the task of managing transactional data.

The term *NoSQL* refers to data stores that do not use SQL for queries. Instead, the data stores use other programming languages and constructs to query the data. In practice, "NoSQL" means "non-relational database," even though many of these databases do support SQL-compatible queries. However, the underlying query execution strategy is usually very different from the way a traditional RDBMS would execute the same SQL query.

The following sections describe the major categories of non-relational or NoSQL database.

Document data stores

A document data store manages a set of named string fields and object data values in an entity that's referred to as a *document*. These data stores typically store data in the form of JSON documents. Each field value could be a scalar item, such as a number, or a compound element, such as a list or a parent-child collection. The data in the fields of a document can be encoded in various ways, including XML, YAML, JSON, BSON, or even stored as plain text. The fields within documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

Typically, a document contains the entire data for an entity. What items constitute an entity are application-specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document might contain information that would be spread across several relational tables in a relational database management system (RDBMS). A document store does not require that all documents have the same structure. This free-form approach provides a great deal of flexibility. For example, applications can store different data in documents in response to a change in business requirements.

Key	Document
1001	`{ "CustomerID": 99, "OrderItems": [ { "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }`
1002	`{ "CustomerID": 220, "OrderItems": [ { "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }`

The application can retrieve documents by using the document key. The key is a unique identifier for the document, which is often hashed, to help distribute data evenly. Some document databases create the document key automatically. Others enable you to specify an attribute of the document to use as the key. The application can also query documents based on the value of one or more fields. Some document databases support indexing to facilitate fast lookup of documents based on one or more indexed fields.

Many document databases support in-place updates, enabling an application to modify the values of specific fields in a document without rewriting the entire document. Read and write operations over multiple fields in a single document are typically atomic.

Relevant Azure service:

- Azure Cosmos DB

Columnar data stores

A columnar or column-family data store organizes data into columns and rows. In its simplest form, a column-family data store can appear very similar to a relational database, at least conceptually. The real power of a column-family database lies in its denormalized approach to structuring sparse data, which stems from the column-oriented approach to storing data.

You can think of a column-family data store as holding tabular data with rows and columns, but the columns are divided into groups known as column families. Each column family holds a set of columns that are logically related and are typically retrieved or manipulated as a unit. Other data that is accessed separately can be stored in separate column families. Within a column family, new columns can be added dynamically, and rows can be sparse (that is, a row doesn't need to have a value for every column).

The following diagram shows an example with two column families, `Identity` and `Contact Info`. The data for a single entity has the same row key in each column family. This structure, where the rows for any given object in a column family can vary dynamically, is an important benefit of the column-family approach, making this form of data store highly suited for storing data with varying schemas.

CustomerID	Column Family: Identity
001	**First name:** Mu Bae **Last name:** Min
002	**First name:** Francisco **Last name:** Vila Nova **Suffix:** Jr.
003	**First name:** Lena **Last name:** Adamcyz **Title:** Dr.

CustomerID	Column Family: Contact Info
001	**Phone number:** 555-0100 **Email:** someone@example.com
002	**Email:** vilanova@contoso.com
003	**Phone number:** 555-0120

Unlike a key/value store or a document database, most column-family databases physically store data in key order, rather than by computing a hash. The row key is considered the primary index and enables key-based access via a specific key or a range of keys. Some implementations allow you to create secondary indexes over specific columns in a column family. Secondary indexes let you retrieve data by columns value, rather than row key.

On disk, all of the columns within a column family are stored together in the same file, with a specific number of rows in each file. With large data sets, this approach creates a performance benefit by reducing the amount of data that needs to be read from disk when only a few columns are queried together at a time.

Read and write operations for a row are typically atomic within a single column family, although some implementations provide atomicity across the entire row, spanning multiple column families.

Relevant Azure service:

- Cosmos DB Cassandra API
- HBase in HDInsight

Key/value data stores

A key/value store is essentially a large hash table. You associate each data value with a unique key, and the key/value store uses this key to store the data by using an appropriate hashing function. The hashing function is selected to provide an even distribution of hashed keys across the data storage.

Most key/value stores only support simple query, insert, and delete operations. To modify a value (either partially or completely), an application must overwrite the existing data for the entire value. In most implementations, reading or writing a single value is an atomic operation. If the value is large, writing may take some time.

An application can store arbitrary data as a set of values, although some key/value stores impose limits on the maximum size of values. The stored values are opaque to the storage system software. Any schema information must be provided and interpreted by the application. Essentially, values are blobs and the key/value store simply retrieves or stores the value by key.

Key/value stores are highly optimized for applications performing simple lookups using the value of the key, or by a range of keys, but are less suitable for systems that need to query data across different tables of keys/values, such as joining data across multiple tables.

Key/value stores are also not optimized for scenarios where querying or filtering by non-key values is important, rather than performing lookups based only on keys. For example, with a relational database, you can find a record by using a WHERE clause to filter the non-key columns, but key/values stores usually do not have this type of lookup capability for values, or if they do, it requires a slow scan of all values.

A single key/value store can be extremely scalable, as the data store can easily distribute data across multiple nodes on separate machines.

Relevant Azure services:

- Azure Cosmos DB Table API
- Azure Cache for Redis
- Azure Table Storage

Graph data stores

A graph data store manages two types of information, nodes and edges. Nodes represent entities, and edges specify the relationships between these entities. Both nodes and edges can have properties that provide information about that node or edge, similar to columns in a table. Edges can also have a direction indicating the nature of the relationship.

The purpose of a graph data store is to allow an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. The following diagram shows an organization's personnel data structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.

This structure makes it straightforward to perform queries such as "Find all employees who report directly or indirectly to Sarah" or "Who works in the same department as John?" For large graphs with lots of entities and relationships, you can perform complex analyses quickly. Many graph databases provide a query language that you can use to traverse a network of relationships efficiently.

Relevant Azure service:

- Azure Cosmos DB Graph API

Time series data stores

Time series data is a set of values organized by time, and a time series data store is optimized for this type of data. Time series data stores must support a very high number of writes, as they typically collect large amounts of data in real time from a large number of sources. Time series data stores are optimized for storing telemetry data. Scenarios include IoT sensors or application/system counters. Updates are rare, and deletes are often done as bulk operations.

timestamp	deviceid	value
2017-01-05T08:00:00.123	1	90.0
2017-01-05T08:00:01.225	2	75.0
2017-01-05T08:01:01.525	2	78.0

Although the records written to a time series database are generally small, there are often a large number of records, and total data size can grow rapidly. Time series data stores also handle out-of-order and late-arriving data, automatic indexing of data points, and optimizations for queries described in terms of windows of time. This last feature enables queries to run across millions of data points and multiple data streams quickly, in order to support time series visualizations, which is a common way that time series data is consumed.

For more information, see Time series solutions

Relevant Azure services:

- Azure Time Series Insights
- OpenTSDB with HBase on HDInsight

Object data stores

Object data stores are optimized for storing and retrieving large binary objects or blobs such as images, text files, video and audio streams, large application data objects and documents, and virtual machine disk images. An object consists of the stored data, some metadata, and a unique ID for accessing the object. Object stores are designed to support files that are individually very large, as well provide large amounts of total storage to manage all files.

path	blob	metadata
/delays/2017/06/01/flights.cscv	0XAABBCCDDEEF...	{created: 2017-06-02}
/delays/2017/06/02/flights.cscv	0XAADDCCDDEEF...	{created: 2017-06-03}
/delays/2017/06/03/flights.cscv	0XAEBBDEDDEEF...	{created: 2017-06-03}

Some object data stores replicate a given blob across multiple server nodes, which enables fast parallel reads. This process, in turn, enables the scale-out querying of data contained in large files, because multiple processes, typically running on different servers, can each query the large data file simultaneously.

One special case of object data stores is the network file share. Using file shares enables files to be accessed across a network using standard networking protocols like server message block (SMB). Given appropriate security and concurrent access control mechanisms, sharing data in this way can enable distributed services to provide highly scalable data access for basic, low-level operations such as simple read and write requests.

Relevant Azure services:

- Azure Blob Storage
- Azure Data Lake Store
- Azure File Storage

External index data stores

External index data stores provide the ability to search for information held in other data stores and services. An external index acts as a secondary index for any data store, and can be used to index massive volumes of data and provide near real-time access to these indexes.

For example, you might have text files stored in a file system. Finding a file by its file path is quick, but searching based on the contents of the file would require a scan of all of the files, which is slow. An external index lets you create secondary search indexes and then quickly find the path to the files that match your criteria. Another example application of an external index is with key/value stores that only index by the key. You can build a secondary index based on the values in the data, and quickly look up the key that uniquely identifies each matched item.

id	search-document
233358	{"name": "Pacific Crest National Scenic Trail", "county": "San Diego", "elevation":1294, "location": {"type": "Point", "coordinates": [–120.802102,49.00021]}}
801970	{"name": "Lewis and Clark National Historic Trail", "county": "Richland", "elevation":584, "location": {"type": "Point", "coordinates": [–104.8546903,48.1264084]}}
1144102	{"name": "Intake Trail", "county": "Umatilla", "elevation":1076, "location": {"type": "Point", "coordinates": [–118.0468873,45.9981939]}}

The indexes are created by running an indexing process. This can be performed using a pull model, triggered by the data store, or using a push model, initiated by application code. Indexes can be multidimensional and may support free-text searches across large volumes of text data.

External index data stores are often used to support full text and web-based search. In these cases, searching can be exact or fuzzy. A fuzzy search finds documents that match a set of terms and calculates how closely they match. Some external indexes also support linguistic analysis that can return matches based on synonyms, genre expansions (for example, matching "dogs" to "pets"), and stemming (for example, searching for "run" also matches "ran" and "running").

Relevant Azure service:

- Azure Search

Typical requirements

Non-relational data stores often use a different storage architecture from that used by relational databases. Specifically, they tend toward having no fixed schema. Also, they tend not to support transactions, or else restrict the scope of transactions, and they generally don't include secondary indexes for scalability reasons.

The following compares the requirements for each of the non-relational data stores:

REQUIREMENT	DOCUMENT DATA	COLUMN-FAMILY DATA	KEY/VALUE DATA	GRAPH DATA
Normalization	Denormalized	Denormalized	Denormalized	Normalized

REQUIREMENT	DOCUMENT DATA	COLUMN-FAMILY DATA	KEY/VALUE DATA	GRAPH DATA
Schema	Schema on read	Column families defined on write, column schema on read	Schema on read	Schema on read
Consistency (across concurrent transactions)	Tunable consistency, document-level guarantees	Column-family–level guarantees	Key-level guarantees	Graph-level guarantees
Atomicity (transaction scope)	Collection	Table	Table	Graph
Locking Strategy	Optimistic (lock free)	Pessimistic (row locks)	Optimistic (ETag)	
Access pattern	Random access	Aggregates on tall/wide data	Random access	Random access
Indexing	Primary and secondary indexes	Primary and secondary indexes	Primary index only	Primary and secondary indexes
Data shape	Document	Tabular with column families containing columns	Key and value	Graph containing edges and vertices
Sparse	Yes	Yes	Yes	No
Wide (lots of columns/attributes)	Yes	Yes	No	No
Datum size	Small (KBs) to medium (low MBs)	Medium (MBs) to Large (low GBs)	Small (KBs)	Small (KBs)
Overall Maximum Scale	Very Large (PBs)	Very Large (PBs)	Very Large (PBs)	Large (TBs)

REQUIREMENT	TIME SERIES DATA	OBJECT DATA	EXTERNAL INDEX DATA
Normalization	Normalized	Denormalized	Denormalized
Schema	Schema on read	Schema on read	Schema on write
Consistency (across concurrent transactions)	N/A	N/A	N/A
Atomicity (transaction scope)	N/A	Object	N/A
Locking Strategy	N/A	Pessimistic (blob locks)	N/A
Access pattern	Random access and aggregation	Sequential access	Random access

REQUIREMENT	TIME SERIES DATA	OBJECT DATA	EXTERNAL INDEX DATA
Indexing	Primary and secondary indexes	Primary index only	N/A
Data shape	Tabular	Blob and metadata	Document
Sparse	No	N/A	No
Wide (lots of columns/attributes)	No	Yes	Yes
Datum size	Small (KBs)	Large (GBs) to Very Large (TBs)	Small (KBs)
Overall Maximum Scale	Large (low TBs)	Very Large (PBs)	Large (low TBs)

Processing free-form text for search

11/19/2021 • 2 minutes to read • Edit Online

To support search, free-form text processing can be performed against documents containing paragraphs of text.

Text search works by constructing a specialized index that is precomputed against a collection of documents. A client application submits a query that contains the search terms. The query returns a result set, consisting of a list of documents sorted by how well each document matches the search criteria. The result set may also include the context in which the document matches the criteria, which enables the application to highlight the matching phrase in the document.

Free-form text processing can produce useful, actionable data from large amounts of noisy text data. The results can give unstructured documents a well-defined and queryable structure.

Challenges

- Processing a collection of free-form text documents is typically computationally intensive, as well as time intensive.
- In order to search free-form text effectively, the search index should support fuzzy search based on terms that have a similar construction. For example, search indexes are built with lemmatization and linguistic stemming, so that queries for "run" will match documents that contain "ran" and "running."

Architecture

In most scenarios, the source text documents are loaded into object storage such as Azure Storage or Azure Data Lake Store, and then indexed using an external search service. In this scenario, source text documents are physically distinct from a resulting search index that's hosted on a search service. An exception is using full text search within SQL Server or Azure SQL Database. In this case, the document data exists internally in tables managed by the database. Once stored, the documents are processed in a batch to create the index.

Technology choices

Options for creating a search index include Azure Cognitive Search, Elasticsearch, and HDInsight with Solr. Each of these technologies can populate a search index from a collection of documents. Cognitive Search provides

indexers that can automatically populate the index for documents ranging from plain text to Excel and PDF formats. You can also attach machine learning models to an indexer to analyze images and unstructured text for searchable content. On HDInsight, Apache Solr can index binary files of many types, including plain text, Word, and PDF. Once the index is constructed, clients can access the search interface by means of a REST API.

If your text data is stored in SQL Server or Azure SQL Database, you can use the full-text search that is built into the database. The database populates the index from text, binary, or XML data stored within the same database. Clients search by using T-SQL queries.

For more information, see Search data stores.

Time series solutions

11/19/2021 • 4 minutes to read • Edit Online

Time series data is a set of values organized by time. Examples of time series data include sensor data, stock prices, click stream data, and application telemetry. Time series data can be analyzed for historical trends, real-time alerts, or predictive modeling.

Time series data represents how an asset or process changes over time. The data has a timestamp, but more importantly, time is the most meaningful axis for viewing or analyzing the data. Time series data typically arrives in order of time and is usually treated as an insert rather than an update to your database. Because of this, change is measured over time, enabling you to look backward and to predict future change. As such, time series data is best visualized with scatter or line charts.

Some examples of time series data are:

- Stock prices captured over time to detect trends.
- Server performance, such as CPU usage, I/O load, memory usage, and network bandwidth consumption.
- Telemetry from sensors on industrial equipment, which can be used to detect pending equipment failure and trigger alert notifications.
- Real-time car telemetry data including speed, braking, and acceleration over a time window to produce an aggregate risk score for the driver.

In each of these cases, you can see how time is most meaningful as an axis. Displaying the events in the order in which they arrived is a key characteristic of time series data, as there is a natural temporal ordering. This differs

from data captured for standard OLTP data pipelines where data can be entered in any order, and updated at any time.

When to use this solution

Choose a time series solution when you need to ingest data whose strategic value is centered around changes over a period of time, and you are primarily inserting new data and rarely updating, if at all. You can use this information to detect anomalies, visualize trends, and compare current data to historical data, among other things. This type of architecture is also best suited for predictive modeling and forecasting results, because you have the historical record of changes over time, which can be applied to any number of forecasting models.

Using time series offers the following benefits:

- Clearly represents how an asset or process changes over time.
- Helps you quickly detect changes to a number of related sources, making anomalies and emerging trends clearly stand out.
- Best suited for predictive modeling and forecasting.

Internet of Things (IoT)

Data collected by IoT devices is a natural fit for time series storage and analysis. The incoming data is inserted and rarely, if ever, updated. The data is timestamped and inserted in the order it was received, and this data is typically displayed in chronological order, enabling users to discover trends, spot anomalies, and use the information for predictive analysis.

For more information, see Internet of Things.

Real-time analytics

Time series data is often time sensitive — that is, it must be acted on quickly, to spot trends in real time or generate alerts. In these scenarios, any delay in insights can cause downtime and business impact. In addition, there is often a need to correlate data from a variety of different sources, such as sensors.

Ideally, you would have a stream processing layer that can handle the incoming data in real time and process all of it with high precision and high granularity. This isn't always possible, depending on your streaming architecture and the components of your stream buffering and stream processing layers. You may need to sacrifice some precision of the time series data by reducing it. This is done by processing sliding time windows (several seconds, for example), allowing the processing layer to perform calculations in a timely manner. You may also need to downsample and aggregate your data when displaying longer periods of time, such as zooming to display data captured over several months.

Challenges

- Time series data is often very high volume, especially in IoT scenarios. Storing, indexing, querying, analyzing, and visualizing time series data can be challenging.
- It can be challenging to find the right combination of high-speed storage and powerful compute operations for handling real-time analytics, while minimizing time to market and overall cost investment.

Architecture

In many scenarios that involve time series data, such as IoT, the data is captured in real time. As such, a real-time processing architecture is appropriate.

Data from one or more data sources is ingested into the stream buffering layer by IoT Hub, Event Hubs, or Kafka on HDInsight. Next, the data is processed in the stream processing layer that can optionally hand off the processed data to a machine learning service for predictive analytics. The processed data is stored in an

analytical data store, such as Azure Data Explorer, HBase, Azure Cosmos DB, Azure Data Lake, or Blob Storage. An analytics and reporting application or service, like Power BI or OpenTSDB (if stored in HBase), can be used to display the time series data for analysis.

Another option is to use Azure Time Series Insights. Time Series Insights is a fully managed service for time series data. In this architecture, Time Series Insights performs the roles of stream processing, data store, and analytics and reporting. It accepts streaming data from either IoT Hub or Event Hubs and stores, processes, analyzes, and displays the data in near real time. It does not pre-aggregate the data, but stores the raw events.

Time Series Insights is schema adaptive, which means that you do not have to do any data preparation to start deriving insights. This enables you to explore, compare, and correlate a variety of data sources seamlessly. It also provides SQL-like filters and aggregates, ability to construct, visualize, compare, and overlay various time series patterns, heat maps, and the ability to save and share queries.

Technology choices

- Data Storage
- Analysis, visualizations, and reporting
- Analytical Data Stores
- Stream processing

Working with CSV and JSON files for data solutions

11/19/2021 • 4 minutes to read • Edit Online

CSV and JSON are likely the most common formats used for ingesting, exchanging, and storing unstructured or semi-structured data.

About CSV format

CSV (comma-separated values) files are commonly used to exchange tabular data between systems in plain text. They typically contain a header row that provides column names for the data, but are otherwise considered semi-structured. This is due to the fact that CSVs cannot naturally represent hierarchical or relational data. Data relationships are typically handled with multiple CSV files, where foreign keys are stored in columns of one or more files, but the relationships between those files are not expressed by the format itself. Files in CSV format may use other delimiters besides commas, such as tabs or spaces.

Despite their limitations, CSV files are a popular choice for data exchange, because they are supported by a wide range of business, consumer, and scientific applications. For example, database and spreadsheet programs can import and export CSV files. Similarly, most batch and stream data processing engines, such as Spark and Hadoop, natively support serializing and deserializing CSV-formatted files and offer ways to apply a schema on read. This makes it easier to work with the data, by offering options to query against it and store the information in a more efficient data format for faster processing.

About JSON format

JSON (JavaScript Object Notation) data is represented as key-value pairs in a semi-structured format. JSON is often compared to XML, as both are capable of storing data in hierarchical format, with child data represented inline with its parent. Both are self-describing and human readable, but JSON documents tend to be much smaller, leading to their popular use in online data exchange, especially with the advent of REST-based web services.

JSON-formatted files have several benefits over CSV:

- JSON maintains hierarchical structures, making it easier to hold related data in a single document and represent complex relationships.
- Most programming languages provide native support for deserializing JSON into objects, or provide lightweight JSON serialization libraries.
- JSON supports lists of objects, helping to avoid messy translations of lists into a relational data model.
- JSON is a commonly used file format for NoSQL databases, such as MongoDB, Couchbase, and Azure Cosmos DB.

Since a lot of data coming across the wire is already in JSON format, most web-based programming languages support working with JSON natively, or through the use of external libraries to serialize and deserialize JSON data. This universal support for JSON has led to its use in logical formats through data structure representation, exchange formats for hot data, and data storage for cold data.

Many batch and stream data processing engines natively support JSON serialization and deserialization. Though the data contained within JSON documents may ultimately be stored in a more performance-optimized formats, such as Parquet or Avro, it serves as the raw data for source of truth, which is critical for reprocessing the data as needed.

When to use CSV or JSON formats

CSVs are more commonly used for exporting and importing data, or processing it for analytics and machine learning. JSON-formatted files have the same benefits, but are more common in hot data exchange solutions. JSON documents are often sent by web and mobile devices performing online transactions, by IoT (internet of things) devices for one-way or bidirectional communication, or by client applications communicating with SaaS and PaaS services or serverless architectures.

CSV and JSON file formats both make it easy to exchange data between dissimilar systems or devices. Their semi-structured formats allow flexibility in transferring almost any type of data, and universal support for these formats make them simple to work with. Both can be used as the raw source of truth in cases where the processed data is stored in binary formats for more efficient querying.

Working with CSV and JSON data in Azure

Azure provides several solutions for working with CSV and JSON files, depending on your needs. The primary landing place for these files is either Azure Storage or Azure Data Lake Store. Most Azure services that work with these and other text-based files integrate with either object storage service. In some situations, however, you may opt to directly import the data into Azure SQL or some other data store. SQL Server has native support for storing and working with JSON documents, which makes it easy to import and process those types of files. You can use a utility like SQL Bulk Import to easily import CSV files.

You can also query JSON files directly from Azure Blob Storage without importing them into Azure SQL. For a complete example of this approach, see Work with JSON files with Azure SQL. Currently this option isn't available for CSV files.

Depending on the scenario, you may perform batch processing or real-time processing of the data.

Challenges

There are some challenges to consider when working with these formats:

- Without any restraints on the data model, CSV and JSON files are prone to data corruption ("garbage in, garbage out"). For instance, there's no notion of a date/time object in either file, so the file format does not prevent you from inserting "ABC123" in a date field, for example.
- Using CSV and JSON files as your cold storage solution does not scale well when working with big data. In most cases, they cannot be split into partitions for parallel processing, and cannot be compressed as well as binary formats. This often leads to processing and storing this data into read-optimized formats such as Parquet and ORC (optimized row columnar), which also provide indexes and inline statistics about the data contained.
- You may need to apply a schema on the semi-structured data to make it easier to query and analyze. Typically, this requires storing the data in another form that complies with your environment's data storage needs, such as within a database.

Build a scalable system for massive data

11/19/2021 • 2 minutes to read • Edit Online

Your data storage system is fundamental to the success of your applications, and therefore to the success of your enterprise. When the storage system is well architected, response is quick, data storage capacity is easily adjusted as necessary, the system is resilient to failures, and it's affordable.

A crucial consideration is whether the design scales well as data grows. As an example of data growth, consider an application that generates 6 terabytes (TB) of data its first month, with the amount increasing every month at a 10 percent yearly rate. Here's a graph that shows how data accumulates over time:

After three years, there's 249 TB of data. If the system is well architected, it handles such data growth gracefully, remaining responsive, resilient, and affordable.

This example isn't extreme. If your customers are businesses, data grows both as you add customers and as your customers add data. It can also grow because of application enhancements.

Handling data growth may require a mix of storage products. For example, you may need to keep rarely accessed data in low-cost services, and frequently accessed data in higher-cost services with better access times.

To design such a system on Azure, you need to be familiar with the many Azure services and with how to use them for various types of applications and various objectives. The articles in this section provide seven system architectures for web applications that use massive amounts of data and that are resilient to system failures. They serve as examples that can help you design a storage system that properly accommodates your applications.

The architectures demonstrate the use of these Azure products: Azure Table Storage, Azure Cosmos DB, Azure Data Factory, and Azure Data Lake.

This capability matrix provides links to the articles and summarizes the benefits and risks of each architecture:

ARCHITECTURE	BENEFITS	RISKS
Two-region web application with Table Storage failover	Straightforward, low-cost implementation	Limited resiliency—only two Azure regions
Multi-region web application with custom Storage Table replication	Resiliency	Implementation time and difficulty
Multi-region web application with Cosmos DB replication	Resiliency, performance, scalability	Storage costs
Optimized storage with logical data classification	Resiliency, performance, scalability, storage costs	Implementation time, need to design logical data classification
Optimized Storage – time based – multi writes	Storage costs	Limited resiliency, performance, limited scalability, implementation time, need to design time-based data retention
Optimized Storage – time based with Data Lake	Resiliency, performance, scalability	Implementation time, need to design time-based data retention
Minimal storage – change feed to replicate data	Resiliency, performance, time-based data retention	Limited scalability, implementation time

Next steps

Here are resources to help you design your storage solution and investigate its business aspects, including costs and service-level agreements.

Design storage solutions

- Build great solutions with the Microsoft Azure Well-Architected Framework
- Understand data store models
- Select an Azure data store for your application
- Criteria for choosing a data store
- Choose a data storage approach in Azure
- Developing with Azure Cosmos DB Table API and Azure Table storage

Azure service limits, cost, service level agreements (SLA), and regional availability

- Azure subscription and service limits, quotas, and constraints
- Azure pricing
- Service-level agreements
- Products available by region

Big data architectures

11/19/2021 • 10 minutes to read • Edit Online

A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems. The threshold at which organizations enter into the big data realm differs, depending on the capabilities of the users and their tools. For some, it can mean hundreds of gigabytes of data, while for others it means hundreds of terabytes. As tools for working with big datasets advance, so does the meaning of big data. More and more, this term relates to the value you can extract from your data sets through advanced analytics, rather than strictly the size of the data, although in these cases they tend to be quite large.

Over the years, the data landscape has changed. What you can do, or are expected to do, with data has changed. The cost of storage has fallen dramatically, while the means by which data is collected keeps growing. Some data arrives at a rapid pace, constantly demanding to be collected and observed. Other data arrives more slowly, but in very large chunks, often in the form of decades of historical data. You might be facing an advanced analytics problem, or one that requires machine learning. These are challenges that big data architectures seek to solve.

Big data solutions typically involve one or more of the following types of workload:

- Batch processing of big data sources at rest.
- Real-time processing of big data in motion.
- Interactive exploration of big data.
- Predictive analytics and machine learning.

Consider big data architectures when you need to:

- Store and process data in volumes too large for a traditional database.
- Transform unstructured data for analysis and reporting.
- Capture, process, and analyze unbounded streams of data in real time, or with low latency.

Components of a big data architecture

The following diagram shows the logical components that fit into a big data architecture. Individual solutions may not contain every item in this diagram.

Most big data architectures include some or all of the following components:

- **Data sources**. All big data solutions start with one or more data sources. Examples include:
  - Application data stores, such as relational databases.
  - Static files produced by applications, such as web server log files.
  - Real-time data sources, such as IoT devices.
- **Data storage**. Data for batch processing operations is typically stored in a distributed file store that can hold high volumes of large files in various formats. This kind of store is often called a *data lake*. Options for implementing this storage include Azure Data Lake Store or blob containers in Azure Storage.
- **Batch processing**. Because the data sets are so large, often a big data solution must process data files using long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. Usually these jobs involve reading source files, processing them, and writing the output to new files. Options include running U-SQL jobs in Azure Data Lake Analytics, using Hive, Pig, or custom Map/Reduce jobs in an HDInsight Hadoop cluster, or using Java, Scala, or Python programs in an HDInsight Spark cluster.
- **Real-time message ingestion**. If the solution includes real-time sources, the architecture must include a way to capture and store real-time messages for stream processing. This might be a simple data store, where incoming messages are dropped into a folder for processing. However, many solutions need a message ingestion store to act as a buffer for messages, and to support scale-out processing, reliable delivery, and other message queuing semantics. This portion of a streaming architecture is often referred to as stream buffering. Options include Azure Event Hubs, Azure IoT Hub, and Kafka.
- **Stream processing**. After capturing real-time messages, the solution must process them by filtering, aggregating, and otherwise preparing the data for analysis. The processed stream data is then written to an output sink. Azure Stream Analytics provides a managed stream processing service based on perpetually running SQL queries that operate on unbounded streams. You can also use open source Apache streaming technologies like Storm and Spark Streaming in an HDInsight cluster.
- **Analytical data store**. Many big data solutions prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools. The analytical data store used to serve these queries can be a Kimball-style relational data warehouse, as seen in most traditional business intelligence (BI) solutions. Alternatively, the data could be presented through a low-latency NoSQL technology such as HBase, or an interactive Hive database that provides a metadata abstraction over data files in the distributed data store. Azure Synapse Analytics provides a managed service for large-scale, cloud-based data warehousing. HDInsight supports Interactive Hive, HBase, and Spark SQL, which can also be used to serve data for analysis.
- **Analysis and reporting**. The goal of most big data solutions is to provide insights into the data through analysis and reporting. To empower users to analyze the data, the architecture may include a data modeling layer, such as a multidimensional OLAP cube or tabular data model in Azure Analysis Services. It might also support self-service BI, using the modeling and visualization technologies in Microsoft Power BI or Microsoft Excel. Analysis and reporting can also take the form of interactive data exploration by data scientists or data analysts. For these scenarios, many Azure services support analytical notebooks, such as Jupyter, enabling these users to leverage their existing skills with Python or R. For large-scale data exploration, you can use Microsoft R Server, either standalone or with Spark.
- **Orchestration**. Most big data solutions consist of repeated data processing operations, encapsulated in workflows, that transform source data, move data between multiple sources and sinks, load the processed data into an analytical data store, or push the results straight to a report or dashboard. To automate these workflows, you can use an orchestration technology such Azure Data Factory or Apache Oozie and Sqoop.

Lambda architecture

When working with very large data sets, it can take a long time to run the sort of queries that clients need. These queries can't be performed in real time, and often require algorithms such as MapReduce that operate in parallel across the entire data set. The results are then stored separately from the raw data and used for querying.

One drawback to this approach is that it introduces latency — if processing takes a few hours, a query may return results that are several hours old. Ideally, you would like to get some results in real time (perhaps with some loss of accuracy), and combine these results with the results from the batch analytics.

The **lambda architecture**, first proposed by Nathan Marz, addresses this problem by creating two paths for data flow. All data coming into the system goes through these two paths:

- A **batch layer** (cold path) stores all of the incoming data in its raw form and performs batch processing on the data. The result of this processing is stored as a **batch view**.
- A **speed layer** (hot path) analyzes data in real time. This layer is designed for low latency, at the expense of accuracy.

The batch layer feeds into a **serving layer** that indexes the batch view for efficient querying. The speed layer updates the serving layer with incremental updates based on the most recent data.

Data that flows into the hot path is constrained by latency requirements imposed by the speed layer, so that it can be processed as quickly as possible. Often, this requires a tradeoff of some level of accuracy in favor of data that is ready as quickly as possible. For example, consider an IoT scenario where a large number of temperature sensors are sending telemetry data. The speed layer may be used to process a sliding time window of the incoming data.

Data flowing into the cold path, on the other hand, is not subject to the same low latency requirements. This allows for high accuracy computation across large data sets, which can be very time intensive.

Eventually, the hot and cold paths converge at the analytics client application. If the client needs to display timely, yet potentially less accurate data in real time, it will acquire its result from the hot path. Otherwise, it will select results from the cold path to display less timely but more accurate data. In other words, the hot path has data for a relatively small window of time, after which the results can be updated with more accurate data from the cold path.

The raw data stored at the batch layer is immutable. Incoming data is always appended to the existing data, and the previous data is never overwritten. Any changes to the value of a particular datum are stored as a new timestamped event record. This allows for recomputation at any point in time across the history of the data collected. The ability to recompute the batch view from the original raw data is important, because it allows for new views to be created as the system evolves.

Kappa architecture

A drawback to the lambda architecture is its complexity. Processing logic appears in two different places — the cold and hot paths — using different frameworks. This leads to duplicate computation logic and the complexity of managing the architecture for both paths.

The **kappa architecture** was proposed by Jay Kreps as an alternative to the lambda architecture. It has the same basic goals as the lambda architecture, but with an important distinction: All data flows through a single path, using a stream processing system.

There are some similarities to the lambda architecture's batch layer, in that the event data is immutable and all of it is collected, instead of a subset. The data is ingested as a stream of events into a distributed and fault tolerant unified log. These events are ordered, and the current state of an event is changed only by a new event being appended. Similar to a lambda architecture's speed layer, all event processing is performed on the input stream and persisted as a real-time view.

If you need to recompute the entire data set (equivalent to what the batch layer does in lambda), you simply replay the stream, typically using parallelism to complete the computation in a timely fashion.

Internet of Things (IoT)

From a practical viewpoint, Internet of Things (IoT) represents any device that is connected to the Internet. This includes your PC, mobile phone, smart watch, smart thermostat, smart refrigerator, connected automobile, heart monitoring implants, and anything else that connects to the Internet and sends or receives data. The number of connected devices grows every day, as does the amount of data collected from them. Often this data is being collected in highly constrained, sometimes high-latency environments. In other cases, data is sent from low-latency environments by thousands or millions of devices, requiring the ability to rapidly ingest the data and process accordingly. Therefore, proper planning is required to handle these constraints and unique requirements.

Event-driven architectures are central to IoT solutions. The following diagram shows a possible logical architecture for IoT. The diagram emphasizes the event-streaming components of the architecture.

The **cloud gateway** ingests device events at the cloud boundary, using a reliable, low latency messaging system.

Devices might send events directly to the cloud gateway, or through a **field gateway**. A field gateway is a specialized device or software, usually collocated with the devices, that receives events and forwards them to the cloud gateway. The field gateway might also preprocess the raw device events, performing functions such as filtering, aggregation, or protocol transformation.

After ingestion, events go through one or more **stream processors** that can route the data (for example, to storage) or perform analytics and other processing.

The following are some common types of processing. (This list is certainly not exhaustive.)

- Writing event data to cold storage, for archiving or batch analytics.
- Hot path analytics, analyzing the event stream in (near) real time, to detect anomalies, recognize patterns over rolling time windows, or trigger alerts when a specific condition occurs in the stream.
- Handling special types of nontelemetry messages from devices, such as notifications and alarms.
- Machine learning.

The boxes that are shaded gray show components of an IoT system that are not directly related to event streaming, but are included here for completeness.

- The **device registry** is a database of the provisioned devices, including the device IDs and usually device metadata, such as location.
- The **provisioning API** is a common external interface for provisioning and registering new devices.
- Some IoT solutions allow **command and control messages** to be sent to devices.

Relevant Azure services:

- Azure IoT Hub
- Azure Event Hubs
- Azure Stream Analytics
- Azure Data Explorer

Learn more about IoT on Azure by reading the Azure IoT reference architecture.

Batch processing

11/19/2021 • 6 minutes to read • Edit Online

A common big data scenario is batch processing of data at rest. In this scenario, the source data is loaded into data storage, either by the source application itself or by an orchestration workflow. The data is then processed in-place by a parallelized job, which can also be initiated by the orchestration workflow. The processing may include multiple iterative steps before the transformed results are loaded into an analytical data store, which can be queried by analytics and reporting components.

For example, the logs from a web server might be copied to a folder and then processed overnight to generate daily reports of web activity.

When to use this solution

Batch processing is used in a variety of scenarios, from simple data transformations to a more complete ETL (extract-transform-load) pipeline. In a big data context, batch processing may operate over very large data sets, where the computation takes significant time. (For example, see Lambda architecture.) Batch processing typically leads to further interactive exploration, provides the modeling-ready data for machine learning, or writes the data to a data store that is optimized for analytics and visualization.

One example of batch processing is transforming a large set of flat, semi-structured CSV or JSON files into a schematized and structured format that is ready for further querying. Typically the data is converted from the raw formats used for ingestion (such as CSV) into binary formats that are more performant for querying because they store data in a columnar format, and often provide indexes and inline statistics about the data.

Challenges

- **Data format and encoding**. Some of the most difficult issues to debug happen when files use an unexpected format or encoding. For example, source files might use a mix of UTF-16 and UTF-8 encoding, or contain unexpected delimiters (space versus tab), or include unexpected characters. Another common example is text fields that contain tabs, spaces, or commas that are interpreted as delimiters. Data loading and parsing logic must be flexible enough to detect and handle these issues.

- **Orchestrating time slices**. Often source data is placed in a folder hierarchy that reflects processing windows, organized by year, month, day, hour, and so on. In some cases, data may arrive late. For example, suppose that a web server fails, and the logs for March 7th don't end up in the folder for processing until March 9th. Are they just ignored because they're too late? Can the downstream

processing logic handle out-of-order records?

Architecture

A batch processing architecture has the following logical components, shown in the diagram above.

- **Data storage**. Typically a distributed file store that can serve as a repository for high volumes of large files in various formats. Generically, this kind of store is often referred to as a data lake.
- **Batch processing**. The high-volume nature of big data often means that solutions must process data files using long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. Usually these jobs involve reading source files, processing them, and writing the output to new files.
- **Analytical data store**. Many big data solutions are designed to prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools.
- **Analysis and reporting**. The goal of most big data solutions is to provide insights into the data through analysis and reporting.
- **Orchestration**. With batch processing, typically some orchestration is required to migrate or copy the data into your data storage, batch processing, analytical data store, and reporting layers.

Technology choices

The following technologies are recommended choices for batch processing solutions in Azure.

Data storage

- **Azure Storage Blob Containers**. Many existing Azure business processes already use Azure blob storage, making this a good choice for a big data store.
- **Azure Data Lake Store**. Azure Data Lake Store offers virtually unlimited storage for any size of file, and extensive security options, making it a good choice for extremely large-scale big data solutions that require a centralized store for data in heterogeneous formats.

For more information, see Data storage.

Batch processing

- **U-SQL**. U-SQL is the query processing language used by Azure Data Lake Analytics. It combines the declarative nature of SQL with the procedural extensibility of C#, and takes advantage of parallelism to enable efficient processing of data at massive scale.
- **Hive**. Hive is a SQL-like language that is supported in most Hadoop distributions, including HDInsight. It can be used to process data from any HDFS-compatible store, including Azure blob storage and Azure Data Lake Store.
- **Pig**. Pig is a declarative big data processing language used in many Hadoop distributions, including HDInsight. It is particularly useful for processing data that is unstructured or semi-structured.
- **Spark**. The Spark engine supports batch processing programs written in a range of languages, including Java, Scala, and Python. Spark uses a distributed architecture to process data in parallel across multiple worker nodes.

For more information, see Batch processing.

Analytical data store

- **Azure Synapse Analytics**. Azure Synapse is a managed service based on SQL Server database technologies and optimized to support large-scale data warehousing workloads.
- **Spark SQL**. Spark SQL is an API built on Spark that supports the creation of dataframes and tables that can be queried using SQL syntax.

- **HBase**. HBase is a low-latency NoSQL store that offers a high-performance, flexible option for querying structured and semi-structured data.
- **Hive**. In addition to being useful for batch processing, Hive offers a database architecture that is conceptually similar to that of a typical relational database management system. Improvements in Hive query performance through innovations like the Tez engine and Stinger initiative mean that Hive tables can be used effectively as sources for analytical queries in some scenarios.

For more information, see Analytical data stores.

Analytics and reporting

- **Azure Analysis Services**. Many big data solutions emulate traditional enterprise business intelligence architectures by including a centralized online analytical processing (OLAP) data model (often referred to as a cube) on which reports, dashboards, and interactive "slice and dice" analysis can be based. Azure Analysis Services supports the creation of tabular models to meet this need.
- **Power BI**. Power BI enables data analysts to create interactive data visualizations based on data models in an OLAP model or directly from an analytical data store.
- **Microsoft Excel**. Microsoft Excel is one of the most widely used software applications in the world, and offers a wealth of data analysis and visualization capabilities. Data analysts can use Excel to build document data models from analytical data stores, or to retrieve data from OLAP data models into interactive PivotTables and charts.

For more information, see Analytics and reporting.

Orchestration

- **Azure Data Factory**. Azure Data Factory pipelines can be used to define a sequence of activities, scheduled for recurring temporal windows. These activities can initiate data copy operations as well as Hive, Pig, MapReduce, or Spark jobs in on-demand HDInsight clusters; U-SQL jobs in Azure Date Lake Analytics; and stored procedures in Azure Synapse or Azure SQL Database.
- **Oozie** and **Sqoop**. Oozie is a job automation engine for the Apache Hadoop ecosystem and can be used to initiate data copy operations as well as Hive, Pig, and MapReduce jobs to process data and Sqoop jobs to copy data between HDFS and SQL databases.

For more information, see Pipeline orchestration

Real time processing

11/19/2021 • 5 minutes to read • Edit Online

Real time processing deals with streams of data that are captured in real-time and processed with minimal latency to generate real-time (or near-real-time) reports or automated responses. For example, a real-time traffic monitoring solution might use sensor data to detect high traffic volumes. This data could be used to dynamically update a map to show congestion, or automatically initiate high-occupancy lanes or other traffic management systems.

Real-time processing is defined as the processing of unbounded stream of input data, with very short latency requirements for processing — measured in milliseconds or seconds. This incoming data typically arrives in an unstructured or semi-structured format, such as JSON, and has the same processing requirements as batch processing, but with shorter turnaround times to support real-time consumption.

Processed data is often written to an analytical data store, which is optimized for analytics and visualization. The processed data can also be ingested directly into the analytics and reporting layer for analysis, business intelligence, and real-time dashboard visualization.

Challenges

One of the big challenges of real-time processing solutions is to ingest, process, and store messages in real time, especially at high volumes. Processing must be done in such a way that it does not block the ingestion pipeline. The data store must support high-volume writes. Another challenge is being able to act on the data quickly, such as generating alerts in real time or presenting the data in a real-time (or near-real-time) dashboard.

Architecture

A real-time processing architecture has the following logical components.

- **Real-time message ingestion.** The architecture must include a way to capture and store real-time messages to be consumed by a stream processing consumer. In simple cases, this service could be implemented as a simple data store in which new messages are deposited in a folder. But often the solution requires a message broker, such as Azure Event Hubs, that acts as a buffer for the messages. The message broker should support scale-out processing and reliable delivery.
- **Stream processing.** After capturing real-time messages, the solution must process them by filtering, aggregating, and otherwise preparing the data for analysis.

- **Analytical data store.** Many big data solutions are designed to prepare data for analysis and then serve the processed data in a structured format that can be queried using analytical tools.
- **Analysis and reporting.** The goal of most big data solutions is to provide insights into the data through analysis and reporting.

Technology choices

The following technologies are recommended choices for real-time processing solutions in Azure.

Real-time message ingestion

- **Azure Event Hubs**. Azure Event Hubs is a messaging solution for ingesting millions of event messages per second. The captured event data can be processed by multiple consumers in parallel. While Event Hubs natively supports AMQP (Advanced Message Queuing Protocol 1.0), it also provides a binary compatibility layer that allows applications using the Kafka protocol (Kafka 1.0 and above) to process events using Event Hubs with no application changes.
- **Azure IoT Hub**. Azure IoT Hub provides bi-directional communication between Internet-connected devices, and a scalable message queue that can handle millions of simultaneously connected devices.
- **Apache Kafka**. Kafka is an open source message queuing and stream processing application that can scale to handle millions of messages per second from multiple message producers, and route them to multiple consumers. Kafka is available in Azure as an HDInsight cluster type, with Azure Events for Kafka, and also available via ConfluentCloud through our partnership with Confluent.

For more information, see Real-time message ingestion.

Data storage

- **Azure Storage Blob Containers** or **Azure Data Lake Store**. Incoming real-time data is usually captured in a message broker (see above), but in some scenarios, it can make sense to monitor a folder for new files and process them as they are created or updated. Additionally, many real-time processing solutions combine streaming data with static reference data, which can be stored in a file store. Finally, file storage may be used as an output destination for captured real-time data for archiving, or for further batch processing in a lambda architecture.

For more information, see Data storage.

Stream processing

- **Azure Stream Analytics**. Azure Stream Analytics can run perpetual queries against an unbounded stream of data. These queries consume streams of data from storage or message brokers, filter and aggregate the data based on temporal windows, and write the results to sinks such as storage, databases, or directly to reports in Power BI. Stream Analytics uses a SQL-based query language that supports temporal and geospatial constructs, and can be extended using JavaScript.
- **Storm**. Apache Storm is an open source framework for stream processing that uses a topology of spouts and bolts to consume, process, and output the results from real-time streaming data sources. You can provision Storm in an Azure HDInsight cluster, and implement a topology in Java or C#.
- **Spark Streaming**. Apache Spark is an open source distributed platform for general data processing. Spark provides the Spark Streaming API, in which you can write code in any supported Spark language, including Java, Scala, and Python. Spark 2.0 introduced the Spark Structured Streaming API, which provides a simpler and more consistent programming model. Spark 2.0 is available in an Azure HDInsight cluster.

For more information, see Stream processing.

Analytical data store

- **Azure Synapse Analytics**, **Azure Data Explorer**, **HBase**, **Spark**, or **Hive**. Processed real-time data can be stored in a relational database such Synapse Analytics, Azure Data Explorer, a NoSQL store such as HBase, or

as files in distributed storage over which Spark or Hive tables can be defined and queried.

For more information, see Analytical data stores.

Analytics and reporting

- **Azure Analysis Services**, **Power BI**, and **Microsoft Excel**. Processed real-time data that is stored in an analytical data store can be used for historical reporting and analysis in the same way as batch processed data. Additionally, Power BI can be used to publish real-time (or near-real-time) reports and visualizations from analytical data sources where latency is sufficiently low, or in some cases directly from the stream processing output.

For more information, see Analytics and reporting.

In a purely real-time solution, most of the processing orchestration is managed by the message ingestion and stream processing components. However, in a lambda architecture that combines batch processing and real-time processing, you may need to use an orchestration framework such as Azure Data Factory or Apache Oozie and Sqoop to manage batch workflows for captured real-time data.

Next steps

The following reference architecture shows an end-to-end stream processing pipeline:

- Stream processing with Azure Stream Analytics

Choose an analytical data store in Azure

11/19/2021 • 5 minutes to read • Edit Online

In a big data architecture, there is often a need for an analytical data store that serves processed data in a structured format that can be queried using analytical tools. Analytical data stores that support querying of both hot-path and cold-path data are collectively referred to as the serving layer, or data serving storage.

The serving layer deals with processed data from both the hot path and cold path. In the lambda architecture, the serving layer is subdivided into a *speed serving* layer, which stores data that has been processed incrementally, and a *batch serving* layer, which contains the batch-processed output. The serving layer requires strong support for random reads with low latency. Data storage for the speed layer should also support random writes, because batch loading data into this store would introduce undesired delays. On the other hand, data storage for the batch layer does not need to support random writes, but batch writes instead.

There is no single best data management choice for all data storage tasks. Different data management solutions are optimized for different tasks. Most real-world cloud apps and big data processes have a variety of data storage requirements and often use a combination of data storage solutions.

What are your options when choosing an analytical data store?

There are several options for data serving storage in Azure, depending on your needs:

- Azure Synapse Analytics
- Azure Synapse Spark pools
- Azure Databricks
- Azure Data Explorer
- Azure SQL Database
- SQL Server in Azure VM
- HBase/Phoenix on HDInsight
- Hive LLAP on HDInsight
- Azure Analysis Services
- Azure Cosmos DB

These options provide various database models that are optimized for different types of tasks:

- Key/value databases hold a single serialized object for each key value. They're good for storing large volumes of data where you want to get one item for a given key value and you don't have to query based on other properties of the item.
- Document databases are key/value databases in which the values are *documents*. A "document" in this context is a collection of named fields and values. The database typically stores the data in a format such as XML, YAML, JSON, or BSON, but may use plain text. Document databases can query on non-key fields and define secondary indexes to make querying more efficient. This makes a document database more suitable for applications that need to retrieve data based on criteria more complex than the value of the document key. For example, you could query on fields such as product ID, customer ID, or customer name.
- Column-family databases are key/value data stores that structure data storage into collections of related columns called column families. For example, a census database might have one group of columns for a person's name (first, middle, last), one group for the person's address, and one group for the person's profile information (data of birth, gender). The database can store each column family in a separate partition, while keeping all of the data for one person related to the same key. An application can read a single column family

without reading through all of the data for an entity.

- Graph databases store information as a collection of objects and relationships. A graph database can efficiently perform queries that traverse the network of objects and the relationships between them. For example, the objects might be employees in a human resources database, and you might want to facilitate queries such as "find all employees who directly or indirectly work for Scott."
- Telemetry and time series databases are an append-only collection of objects. Telemetry databases efficiently index data in a variety of column stores and in-memory structures, making them the optimal choice for storing and analyzing vast quantities of telemetry and time series data.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need serving storage that can serve as a hot path for your data? If yes, narrow your options to those that are optimized for a speed serving layer.
- Do you need massively parallel processing (MPP) support, where queries are automatically distributed across several processes or nodes? If yes, select an option that supports query scale out.
- Do you prefer to use a relational data store? If so, narrow your options to those with a relational database model. However, note that some non-relational stores support SQL syntax for querying, and tools such as PolyBase can be used to query non-relational data stores.
- Do you collect time series data? Do you use append-only data?

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Is managed service	Yes	Yes	Yes	Yes	Yes [1]	Yes [1]	Yes	Yes
Primary database model	Relational (columnar format when using columnstore indexes)	Relational tables with columnar storage	Wide column store	Relational (column store), telemetry, and time series store	Wide column store	Hive/In-Memory	Tabular semantic models	Document store, graph, key-value store, wide column store
SQL language support	Yes	Yes	Yes	Yes	Yes (using Phoenix JDBC driver)	Yes	No	Yes

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Optimized for speed serving layer	Yes [2]	Yes [3]	Yes	Yes	Yes	Yes	No	Yes

[1] With manual configuration and scaling.

[2] Using memory-optimized tables and hash or nonclustered indexes.

[3] Supported as an Azure Stream Analytics output.

Scalability capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE SQL POOL	AZURE SYNAPSE SPARK POOL	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Redundant regional servers for high availability	Yes	No	No	Yes	Yes	No	No	Yes
Supports query scale out	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic scalability (scale up)	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Supports in-memory caching of data	Yes	Yes	Yes	Yes	No	Yes	Yes	No

Security capabilities

CAPABILITY	SQL DATABASE	AZURE SYNAPSE	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Authentication	SQL / Azure Active Directory (Azure AD)	SQL / Azure AD	Azure AD	local / Azure AD [1]	local / Azure AD [1]	Azure AD	database users / Azure AD via access control (IAM)

CAPABILITY	SQL DATABASE	AZURE SYNAPSE	AZURE DATA EXPLORER	HBASE/PHOENIX ON HDINSIGHT	HIVE LLAP ON HDINSIGHT	AZURE ANALYSIS SERVICES	COSMOS DB
Data encryption at rest	Yes [2]	Yes [2]	Yes	Yes [1]	Yes [1]	Yes	Yes
Row-level security	Yes	Yes [3]	No	Yes [1]	Yes [1]	Yes	No
Supports firewalls	Yes	Yes	Yes	Yes [4]	Yes [4]	Yes	Yes
Dynamic data masking	Yes	Yes	Yes	Yes [1]	Yes	No	No

[1] Requires using a domain-joined HDInsight cluster.

[2] Requires using transparent data encryption (TDE) to encrypt and decrypt your data at rest.

[3] Filter predicates only. See Row-Level Security

[4] When used within an Azure Virtual Network. See Extend Azure HDInsight using an Azure Virtual Network.

Choose a data analytics technology in Azure

11/19/2021 • 4 minutes to read • Edit Online

The goal of most big data solutions is to provide insights into the data through analysis and reporting. This can include preconfigured reports and visualizations, or interactive data exploration.

What are your options when choosing a data analytics technology?

There are several options for analysis, visualizations, and reporting in Azure, depending on your needs:

- Power BI
- Jupyter Notebooks
- Zeppelin Notebooks
- Microsoft Azure Notebooks

Power BI

Power BI is a suite of business analytics tools. It can connect to hundreds of data sources, and can be used for ad hoc analysis. See this list of the currently available data sources. Use Power BI Embedded to integrate Power BI within your own applications without requiring any additional licensing.

Organizations can use Power BI to produce reports and publish them to the organization. Everyone can create personalized dashboards, with governance and security built in. Power BI uses Azure Active Directory (Azure AD) to authenticate users who log in to the Power BI service, and uses the Power BI login credentials whenever a user attempts to access resources that require authentication.

Jupyter Notebooks

Jupyter Notebooks provide a browser-based shell that lets data scientists create *notebook* files that contain Python, Scala, or R code and markdown text, making it an effective way to collaborate by sharing and documenting code and results in a single document.

Most varieties of HDInsight clusters, such as Spark or Hadoop, come preconfigured with Jupyter notebooks for interacting with data and submitting jobs for processing. Depending on the type of HDInsight cluster you are using, one or more kernels will be provided for interpreting and running your code. For example, Spark clusters on HDInsight provide Spark-related kernels that you can select from to execute Python or Scala code using the Spark engine.

Jupyter notebooks provide a great environment for analyzing, visualizing, and processing your data prior to building more advanced visualizations with a BI/reporting tool like Power BI.

Zeppelin Notebooks

Zeppelin Notebooks are another option for a browser-based shell, similar to Jupyter in functionality. Some HDInsight clusters come preconfigured with Zeppelin notebooks. However, if you are using an HDInsight Interactive Query (Hive LLAP) cluster, Zeppelin is currently your only choice of notebook that you can use to run interactive Hive queries. Also, if you are using a domain-joined HDInsight cluster, Zeppelin notebooks are the only type that enables you to assign different user logins to control access to notebooks and the underlying Hive tables.

Microsoft Azure Notebooks

Azure Notebooks is an online Jupyter Notebooks-based service that enables data scientists to create, run, and share Jupyter Notebooks in cloud-based libraries. Azure Notebooks provides execution environments for Python 2, Python 3, F#, and R, and provides several charting libraries for visualizing your data, such as ggplot,

matplotlib, bokeh, and seaborn.

Unlike Jupyter notebooks running on an HDInsight cluster, which are connected to the cluster's default storage account, Azure Notebooks does not provide any data. You must load data in a variety of ways, such downloading data from an online source, interacting with Azure Blobs or Table Storage, connecting to a SQL database, or loading data with the Copy Wizard for Azure Data Factory.

Key benefits:

- Free service—no Azure subscription required.
- No need to install Jupyter and the supporting R or Python distributions locally—just use a browser.
- Manage your own online libraries and access them from any device.
- Share your notebooks with collaborators.

Considerations:

- You will be unable to access your notebooks when offline.
- Limited processing capabilities of the free notebook service may not be enough to train large or complex models.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need to connect to numerous data sources, providing a centralized place to create reports for data spread throughout your domain? If so, choose an option that allows you to connect to 100s of data sources.
- Do you want to embed dynamic visualizations in an external website or application? If so, choose an option that provides embedding capabilities.
- Do you want to design your visualizations and reports while offline? If yes, choose an option with offline capabilities.
- Do you need heavy processing power to train large or complex AI models or work with very large data sets? If yes, choose an option that can connect to a big data cluster.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	POWER BI	JUPYTER NOTEBOOKS	ZEPPELIN NOTEBOOKS	MICROSOFT AZURE NOTEBOOKS
Connect to big data cluster for advanced processing	Yes	Yes	Yes	No
Managed service	Yes	Yes [1]	Yes [1]	Yes
Connect to 100s of data sources	Yes	No	No	No
Offline capabilities	Yes [2]	No	No	No

CAPABILITY	POWER BI	JUPYTER NOTEBOOKS	ZEPPELIN NOTEBOOKS	MICROSOFT AZURE NOTEBOOKS
Embedding capabilities	Yes	No	No	No
Automatic data refresh	Yes	No	No	No
Access to numerous open source packages	No	Yes [3]	Yes [3]	Yes [4]
Data transformation/cleansing options	Power Query, R	40 languages, including Python, R, Julia, and Scala	20+ interpreters, including Python, JDBC, and R	Python, F#, R
Pricing	Free for Power BI Desktop (authoring), see pricing for hosting options	Free	Free	Free
Multiuser collaboration	Yes	Yes (through sharing or with a multiuser server like JupyterHub)	Yes	Yes (through sharing)

[1] When used as part of a managed HDInsight cluster.

[2] With the use of Power BI Desktop.

[2] You can search the Maven repository for community-contributed packages.

[3] Python packages can be installed using either pip or conda. R packages can be installed from CRAN or GitHub. Packages in F# can be installed via nuget.org using the Paket dependency manager.

Choose a batch processing technology in Azure

11/19/2021 • 3 minutes to read • Edit Online

Big data solutions often use long-running batch jobs to filter, aggregate, and otherwise prepare the data for analysis. Usually these jobs involve reading source files from scalable storage (like HDFS, Azure Data Lake Store, and Azure Storage), processing them, and writing the output to new files in scalable storage.

The key requirement of such batch processing engines is the ability to scale out computations, in order to handle a large volume of data. Unlike real-time processing, however, batch processing is expected to have latencies (the time between data ingestion and computing a result) that measure in minutes to hours.

Technology choices for batch processing

Azure Synapse Analytics

Azure Synapse is a distributed system designed to perform analytics on large data. It supports massive parallel processing (MPP), which makes it suitable for running high-performance analytics. Consider Azure Synapse when you have large amounts of data (more than 1 TB) and are running an analytics workload that will benefit from parallelism.

Azure Data Lake Analytics

Data Lake Analytics is an on-demand analytics job service. It is optimized for distributed processing of very large data sets stored in Azure Data Lake Store.

- Languages: U-SQL (including Python, R, and C# extensions).
- Integrates with Azure Data Lake Store, Azure Storage blobs, Azure SQL Database, and Azure Synapse.
- Pricing model is per-job.

HDInsight

HDInsight is a managed Hadoop service. Use it deploy and manage Hadoop clusters in Azure. For batch processing, you can use Spark, Hive, Hive LLAP, MapReduce.

- Languages: R, Python, Java, Scala, SQL
- Kerberos authentication with Active Directory, Apache Ranger based access control
- Gives you full control of the Hadoop cluster

Azure Databricks

Azure Databricks is an Apache Spark-based analytics platform. You can think of it as "Spark as a service." It's the easiest way to use Spark on the Azure platform.

- Languages: R, Python, Java, Scala, Spark SQL
- Fast cluster start times, autotermination, autoscaling.
- Manages the Spark cluster for you.
- Built-in integration with Azure Blob Storage, Azure Data Lake Storage (ADLS), Azure Synapse, and other services. See Data Sources.
- User authentication with Azure Active Directory.
- Web-based notebooks for collaboration and data exploration.
- Supports GPU-enabled clusters

Azure Distributed Data Engineering Toolkit

The Distributed Data Engineering Toolkit (AZTK) is a tool for provisioning on-demand Spark on Docker clusters

in Azure.

AZTK is not an Azure service. Rather, it's a client-side tool with a CLI and Python SDK interface, that's built on Azure Batch. This option gives you the most control over the infrastructure when deploying a Spark cluster.

- Bring your own Docker image.
- Use low-priority VMs for an 80% discount.
- Mixed mode clusters that use both low-priority and dedicated VMs.
- Built in support for Azure Blob Storage and Azure Data Lake connection.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Do you want to author batch processing logic declaratively or imperatively?
- Will you perform batch processing in bursts? If yes, consider options that let you auto-terminate the cluster or whose pricing model is per batch job.
- Do you need to query relational data stores along with your batch processing, for example to look up reference data? If yes, consider the options that enable querying of external relational stores.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE DATA LAKE ANALYTICS	AZURE SYNAPSE	HDINSIGHT	AZURE DATABRICKS
Is managed service	Yes	Yes	Yes [1]	Yes
Relational data store	Yes	Yes	No	No
Pricing model	Per batch job	By cluster hour	By cluster hour	Databricks Unit[2] + cluster hour

[1] With manual configuration.

[2] A Databricks Unit (DBU) is a unit of processing capability per hour.

Capabilities

CAPABILITY	AZURE DATA LAKE ANALYTICS	AZURE SYNAPSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP	AZURE DATABRICKS
Autoscaling	No	No	Yes	Yes	Yes	Yes
Scale-out granularity	Per job	Per cluster	Per cluster	Per cluster	Per cluster	Per cluster
In-memory caching of data	No	Yes	Yes	No	Yes	Yes

CAPABILITY	AZURE DATA LAKE ANALYTICS	AZURE SYNAPSE	HDINSIGHT WITH SPARK	HDINSIGHT WITH HIVE	HDINSIGHT WITH HIVE LLAP	AZURE DATABRICKS
Query from external relational stores	Yes	No	Yes	No	No	Yes
Authentication	Azure AD	SQL / Azure AD	No	Azure AD[1]	Azure AD[1]	Azure AD
Auditing	Yes	Yes	No	Yes [1]	Yes [1]	Yes
Row-level security	No	Yes[2]	No	Yes [1]	Yes [1]	No
Supports firewalls	Yes	Yes	Yes	Yes [3]	Yes [3]	No
Dynamic data masking	No	Yes	No	Yes [1]	Yes [1]	No

[1] Requires using a domain-joined HDInsight cluster.

[2] Filter predicates only. See Row-Level Security

[3] Supported when used within an Azure Virtual Network.

Next steps

- Analytics architecture design
- Choose an analytical data store in Azure
- Choose a data analytics technology in Azure
- Analytics end-to-end with Azure Synapse

Data lakes

11/19/2021 • 2 minutes to read • Edit Online

A data lake is a storage repository that holds a large amount of data in its native, raw format. Data lake stores are optimized for scaling to terabytes and petabytes of data. The data typically comes from multiple heterogeneous sources, and may be structured, semi-structured, or unstructured. The idea with a data lake is to store everything in its original, untransformed state. This approach differs from a traditional data warehouse, which transforms and processes the data at the time of ingestion.

Advantages of a data lake:

- Data is never thrown away, because the data is stored in its raw format. This is especially useful in a big data environment, when you may not know in advance what insights are available from the data.
- Users can explore the data and create their own queries.
- May be faster than traditional ETL tools.
- More flexible than a data warehouse, because it can store unstructured and semi-structured data.

A complete data lake solution consists of both storage and processing. Data lake storage is designed for fault-tolerance, infinite scalability, and high-throughput ingestion of data with varying shapes and sizes. Data lake processing involves one or more processing engines built with these goals in mind, and can operate on data stored in a data lake at scale.

When to use a data lake

Typical uses for a data lake include data exploration, data analytics, and machine learning.

A data lake can also act as the data source for a data warehouse. With this approach, the raw data is ingested into the data lake and then transformed into a structured queryable format. Typically this transformation uses an ELT (extract-load-transform) pipeline, where the data is ingested and transformed in place. Source data that is already relational may go directly into the data warehouse, using an ETL process, skipping the data lake.

Data lake stores are often used in event streaming or IoT scenarios, because they can persist large amounts of relational and nonrelational data without transformation or schema definition. They are built to handle high volumes of small writes at low latency, and are optimized for massive throughput.

Challenges

- Lack of a schema or descriptive metadata can make the data hard to consume or query.
- Lack of semantic consistency across the data can make it challenging to perform analysis on the data, unless users are highly skilled at data analytics.
- It can be hard to guarantee the quality of the data going into the data lake.
- Without proper governance, access control and privacy issues can be problems. What information is going into the data lake, who can access that data, and for what uses?
- A data lake may not be the best way to integrate data that is already relational.
- By itself, a data lake does not provide integrated or holistic views across the organization.
- A data lake may become a dumping ground for data that is never actually analyzed or mined for insights.

Relevant Azure services

- Data Lake Store is a hyperscale, Hadoop-compatible repository.

- Data Lake Analytics is an on-demand analytics job service to simplify big data analytics.

Choose a big data storage technology in Azure

11/19/2021 • 8 minutes to read • Edit Online

This topic compares options for data storage for big data solutions — specifically, data storage for bulk data ingestion and batch processing, as opposed to analytical data stores or real-time streaming ingestion.

What are your options when choosing data storage in Azure?

There are several options for ingesting data into Azure, depending on your needs.

File storage:

- Azure Storage blobs
- Azure Data Lake Store

NoSQL databases:

- Azure Cosmos DB
- HBase on HDInsight

Analytical databases:

Azure Data Explorer

Azure Storage blobs

Azure Storage is a managed storage service that is highly available, secure, durable, scalable, and redundant. Microsoft takes care of maintenance and handles critical problems for you. Azure Storage is the most ubiquitous storage solution Azure provides, due to the number of services and tools that can be used with it.

There are various Azure Storage services you can use to store data. The most flexible option for storing blobs from a number of data sources is Blob storage. Blobs are basically files. They store pictures, documents, HTML files, virtual hard disks (VHDs), big data such as logs, database backups — pretty much anything. Blobs are stored in containers, which are similar to folders. A container provides a grouping of a set of blobs. A storage account can contain an unlimited number of containers, and a container can store an unlimited number of blobs.

Azure Storage is a good choice for big data and analytics solutions, because of its flexibility, high availability, and low cost. It provides hot, cool, and archive storage tiers for different use cases. For more information, see Azure Blob Storage: Hot, cool, and archive storage tiers.

Azure Blob storage can be accessed from Hadoop (available through HDInsight). HDInsight can use a blob container in Azure Storage as the default file system for the cluster. Through a Hadoop distributed file system (HDFS) interface provided by a WASB driver, the full set of components in HDInsight can operate directly on structured or unstructured data stored as blobs. Azure Blob storage can also be accessed via Azure Synapse Analytics using its PolyBase feature.

Other features that make Azure Storage a good choice are:

- Multiple concurrency strategies.
- Disaster recovery and high availability options.
- Encryption at rest.
- Azure role-based access control (Azure RBAC) to control access using Azure Active Directory users and groups.

Azure Data Lake Store

Azure Data Lake Store is an enterprise-wide hyperscale repository for big data analytic workloads. Data Lake enables you to capture data of any size, type, and ingestion speed in one single secure location for operational and exploratory analytics.

Data Lake Store does not impose any limits on account sizes, file sizes, or the amount of data that can be stored in a data lake. Data is stored durably by making multiple copies and there is no limit on the duration of time that the data can be stored in the Data Lake. In addition to making multiple copies of files to guard against any unexpected failures, Data lake spreads parts of a file over a number of individual storage servers. This improves the read throughput when reading the file in parallel for performing data analytics.

Data Lake Store can be accessed from Hadoop (available through HDInsight) using the WebHDFS-compatible REST APIs. You may consider using this as an alternative to Azure Storage when your individual or combined file sizes exceed that which is supported by Azure Storage. However, there are performance tuning guidelines you should follow when using Data Lake Store as your primary storage for an HDInsight cluster, with specific guidelines for Spark, Hive, MapReduce, and Storm. Also, be sure to check Data Lake Store's regional availability, because it is not available in as many regions as Azure Storage, and it needs to be located in the same region as your HDInsight cluster.

Coupled with Azure Data Lake Analytics, Data Lake Store is specifically designed to enable analytics on the stored data and is tuned for performance for data analytics scenarios. Data Lake Store can also be accessed via Azure Synapse using its PolyBase feature.

Azure Cosmos DB

Azure Cosmos DB is Microsoft's globally distributed multi-model database. Cosmos DB guarantees single-digit-millisecond latencies at the 99th percentile anywhere in the world, offers multiple well-defined consistency models to fine-tune performance, and guarantees high availability with multi-homing capabilities.

Azure Cosmos DB is schema-agnostic. It automatically indexes all the data without requiring you to deal with schema and index management. It's also multi-model, natively supporting document, key-value, graph, and column-family data models.

Azure Cosmos DB features:

- Geo-replication
- Elastic scaling of throughput and storage worldwide
- Five well-defined consistency levels

HBase on HDInsight

Apache HBase is an open-source, NoSQL database that is built on Hadoop and modeled after Google BigTable. HBase provides random access and strong consistency for large amounts of unstructured and semi-structured data in a schemaless database organized by column families.

Data is stored in the rows of a table, and data within a row is grouped by column family. HBase is schemaless in the sense that neither the columns nor the type of data stored in them need to be defined before using them. The open-source code scales linearly to handle petabytes of data on thousands of nodes. It can rely on data redundancy, batch processing, and other features that are provided by distributed applications in the Hadoop ecosystem.

The HDInsight implementation leverages the scale-out architecture of HBase to provide automatic sharding of tables, strong consistency for reads and writes, and automatic failover. Performance is enhanced by in-memory caching for reads and high-throughput streaming for writes. In most cases, you'll want to create the HBase cluster inside a virtual network so other HDInsight clusters and applications can directly access the tables.

Azure Data Explorer

Azure Data Explorer is a fast and highly scalable data exploration service for log and telemetry data. It helps you handle the many data streams emitted by modern software so you can collect, store, and analyze data. Azure Data Explorer is ideal for analyzing large volumes of diverse data from any data source, such as websites, applications, IoT devices, and more. This data is used for diagnostics, monitoring, reporting, machine learning, and additional analytics capabilities. Azure Data Explorer makes it simple to ingest this data and enables you to do complex ad hoc queries on the data in seconds.

Azure Data Explorer can be linearly scaled out for increasing ingestion and query processing throughput. An Azure Data Explorer cluster can be deployed to a Virtual Network for enabling private networks.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need managed, high-speed, cloud-based storage for any type of text or binary data? If yes, then select one of the file storage or analytics options.
- Do you need file storage that is optimized for parallel analytics workloads and high throughput/IOPS? If yes, then choose an option that is tuned to analytics workload performance.
- Do you need to store unstructured or semi-structured data in a schemaless database? If so, select one of the non-relational or analytics options. Compare options for indexing and database models. Depending on the type of data you need to store, the primary database models may be the largest factor.
- Can you use the service in your region? Check the regional availability for each Azure service. See Products available by region.

Capability matrix

The following tables summarize the key differences in capabilities.

File storage capabilities

CAPABILITY	AZURE DATA LAKE STORE	AZURE BLOB STORAGE CONTAINERS
Purpose	Optimized storage for big data analytics workloads	General purpose object store for a wide variety of storage scenarios
Use cases	Batch, streaming analytics, and machine learning data such as log files, IoT data, click streams, large datasets	Any type of text or binary data, such as application back end, backup data, media storage for streaming, and general purpose data
Structure	Hierarchical file system	Object store with flat namespace
Authentication	Based on Azure Active Directory Identities	Based on shared secrets Account Access Keys and Shared Access Signature Keys, and Azure role-based access control (Azure RBAC)
Authentication protocol	OAuth 2.0. Calls must contain a valid JWT (JSON web token) issued by Azure Active Directory	Hash-based message authentication code (HMAC). Calls must contain a Base64-encoded SHA-256 hash over a part of the HTTP request.

CAPABILITY	AZURE DATA LAKE STORE	AZURE BLOB STORAGE CONTAINERS
Authorization	POSIX access control lists (ACLs). ACLs based on Azure Active Directory identities can be set file and folder level.	For account-level authorization use Account Access Keys. For account, container, or blob authorization use Shared Access Signature Keys.
Auditing	Available.	Available
Encryption at rest	Transparent, server side	Transparent, server side; Client-side encryption
Developer SDKs	.NET, Java, Python, Node.js	.NET, Java, Python, Node.js, C++, Ruby
Analytics workload performance	Optimized performance for parallel analytics workloads, High Throughput and IOPS	Not optimized for analytics workloads
Size limits	No limits on account sizes, file sizes or number of files	Specific limits documented here
Geo-redundancy	Locally-redundant (LRS), globally redundant (GRS), read-access globally redundant (RA-GRS), zone-redundant (ZRS).	Locally redundant (LRS), globally redundant (GRS), read-access globally redundant (RA-GRS), zone-redundant (ZRS). See here for more information

NoSQL database capabilities

CAPABILITY	AZURE COSMOS DB	HBASE ON HDINSIGHT
Primary database model	Document store, graph, key-value store, wide column store	Wide column store
Secondary indexes	Yes	No
SQL language support	Yes	Yes (using the Phoenix JDBC driver)
Consistency	Strong, bounded-staleness, session, consistent prefix, eventual	Strong
Native Azure Functions integration	Yes	No
Automatic global distribution	Yes	No HBase cluster replication can be configured across regions with eventual consistency
Pricing model	Elastically scalable request units (RUs) charged per-second as needed, elastically scalable storage	Per-minute pricing for HDInsight cluster (horizontal scaling of nodes), storage

Analytical database capabilities

CAPABILITY	AZURE DATA EXPLORER	

CAPABILITY	AZURE DATA EXPLORER	
Primary database model	Relational (column store), telemetry, and time series store	
SQL language support	Yes	
Pricing model	Elastically scalable cluster instances	
Authentication	Based on Azure Active Directory identities	
Encryption at rest	Supported, customer managed keys	
Analytics workload performance	Optimized performance for parallel analytics workloads	
Size limits	Linearly scalable	

Understand data store models

11/19/2021 • 12 minutes to read • Edit Online

Modern business systems manage increasingly large volumes of heterogeneous data. This heterogeneity means that a single data store is usually not the best approach. Instead, it's often better to store different types of data in different data stores, each focused toward a specific workload or usage pattern. The term *polyglot persistence* is used to describe solutions that use a mix of data store technologies. Therefore, it's important to understand the main storage models and their tradeoffs.

Selecting the right data store for your requirements is a key design decision. There are literally hundreds of implementations to choose from among SQL and NoSQL databases. Data stores are often categorized by how they structure data and the types of operations they support. This article describes several of the most common storage models. Note that a particular data store technology may support multiple storage models. For example, a relational database management systems (RDBMS) may also support key/value or graph storage. In fact, there is a general trend for so-called *multi-model* support, where a single database system supports several models. But it's still useful to understand the different models at a high level.

Not all data stores in a given category provide the same feature-set. Most data stores provide server-side functionality to query and process data. Sometimes this functionality is built into the data storage engine. In other cases, the data storage and processing capabilities are separated, and there may be several options for processing and analysis. Data stores also support different programmatic and management interfaces.

Generally, you should start by considering which storage model is best suited for your requirements. Then consider a particular data store within that category, based on factors such as feature set, cost, and ease of management.

> **NOTE**
>
> Learn more about identifying and reviewing your data service requirements for cloud adoption, in the Microsoft Cloud Adoption Framework for Azure. Likewise, you can also learn about selecting storage tools and services.

Relational database management systems

Relational databases organize data as a series of two-dimensional tables with rows and columns. Most vendors provide a dialect of the Structured Query Language (SQL) for retrieving and managing data. An RDBMS typically implements a transactionally consistent mechanism that conforms to the ACID (Atomic, Consistent, Isolated, Durable) model for updating information.

An RDBMS typically supports a schema-on-write model, where the data structure is defined ahead of time, and all read or write operations must use the schema.

This model is very useful when strong consistency guarantees are important — where all changes are atomic, and transactions always leave the data in a consistent state. However, an RDBMS generally can't scale out horizontally without sharding the data in some way. Also, the data in an RDBMS must be normalized, which isn't appropriate for every data set.

Azure services

- Azure SQL Database | (Security Baseline)
- Azure Database for MySQL | (Security Baseline)
- Azure Database for PostgreSQL | (Security Baseline)

- Azure Database for MariaDB | (Security Baseline)

Workload

- Records are frequently created and updated.
- Multiple operations have to be completed in a single transaction.
- Relationships are enforced using database constraints.
- Indexes are used to optimize query performance.

Data type

- Data is highly normalized.
- Database schemas are required and enforced.
- Many-to-many relationships between data entities in the database.
- Constraints are defined in the schema and imposed on any data in the database.
- Data requires high integrity. Indexes and relationships need to be maintained accurately.
- Data requires strong consistency. Transactions operate in a way that ensures all data are 100% consistent for all users and processes.
- Size of individual data entries is small to medium-sized.

Examples

- Inventory management
- Order management
- Reporting database
- Accounting

Key/value stores

A key/value store associates each data value with a unique key. Most key/value stores only support simple query, insert, and delete operations. To modify a value (either partially or completely), an application must overwrite the existing data for the entire value. In most implementations, reading or writing a single value is an atomic operation.

An application can store arbitrary data as a set of values. Any schema information must be provided by the application. The key/value store simply retrieves or stores the value by key.

Key/value stores are highly optimized for applications performing simple lookups, but are less suitable if you need to query data across different key/value stores. Key/value stores are also not optimized for querying by value.

A single key/value store can be extremely scalable, as the data store can easily distribute data across multiple nodes on separate machines.

Azure services

- Azure Cosmos DB Table API, etcd API (preview), and SQL API | (Cosmos DB Security Baseline)
- Azure Cache for Redis | (Security Baseline)
- Azure Table Storage | (Security Baseline)

Workload

- Data is accessed using a single key, like a dictionary.
- No joins, lock, or unions are required.
- No aggregation mechanisms are used.
- Secondary indexes are generally not used.

Data type

- Each key is associated with a single value.
- There is no schema enforcement.
- No relationships between entities.

Examples

- Data caching
- Session management
- User preference and profile management
- Product recommendation and ad serving

Document databases

A document database stores a collection of *documents*, where each document consists of named fields and data. The data can be simple values or complex elements such as lists and child collections. Documents are retrieved by unique keys.

Typically, a document contains the data for single entity, such as a customer or an order. A document may contain information that would be spread across several relational tables in an RDBMS. Documents don't need to have the same structure. Applications can store different data in documents as business requirements change.

Key	Document
1001	{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }
1002	{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }

Azure service

- Azure Cosmos DB SQL API | (Cosmos DB Security Baseline)

Workload

- Insert and update operations are common.
- No object-relational impedance mismatch. Documents can better match the object structures used in application code.
- Individual documents are retrieved and written as a single block.

- Data requires index on multiple fields.

Data type

- Data can be managed in de-normalized way.
- Size of individual document data is relatively small.
- Each document type can use its own schema.
- Documents can include optional fields.
- Document data is semi-structured, meaning that data types of each field are not strictly defined.

Examples

- Product catalog
- Content management
- Inventory management

Graph databases

A graph database stores two types of information, nodes and edges. Edges specify relationships between nodes. Nodes and edges can have properties that provide information about that node or edge, similar to columns in a table. Edges can also have a direction indicating the nature of the relationship.

Graph databases can efficiently perform queries across the network of nodes and edges and analyze the relationships between entities. The following diagram shows an organization's personnel database structured as a graph. The entities are employees and departments, and the edges indicate reporting relationships and the departments in which employees work.

This structure makes it straightforward to perform queries such as "Find all employees who report directly or indirectly to Sarah" or "Who works in the same department as John?" For large graphs with lots of entities and relationships, you can perform very complex analyses very quickly. Many graph databases provide a query language that you can use to traverse a network of relationships efficiently.

Azure services

- Azure Cosmos DB Gremlin API | (Security Baseline)
- SQL Server | (Security Baseline)

Workload

- Complex relationships between data items involving many hops between related data items.
- The relationship between data items are dynamic and change over time.
- Relationships between objects are first-class citizens, without requiring foreign-keys and joins to traverse.

Data type

- Nodes and relationships.
- Nodes are similar to table rows or JSON documents.
- Relationships are just as important as nodes, and are exposed directly in the query language.
- Composite objects, such as a person with multiple phone numbers, tend to be broken into separate, smaller nodes, combined with traversable relationships

Examples

- Organization charts
- Social graphs
- Fraud detection
- Recommendation engines

Data analytics

Data analytics stores provide massively parallel solutions for ingesting, storing, and analyzing data. The data is distributed across multiple servers to maximize scalability. Large data file formats such as delimiter files (CSV), parquet, and ORC are widely used in data analytics. Historical data is typically stored in data stores such as blob storage or Azure Data Lake Storage Gen2, which are then accessed by Azure Synapse, Databricks, or HDInsight as external tables. A typical scenario using data stored as parquet files for performance, is described in the article Use external tables with Synapse SQL.

Azure services

- Azure Synapse Analytics | (Security Baseline)
- Azure Data Lake | (Security Baseline)
- Azure Data Explorer | (Security Baseline)
- Azure Analysis Services
- HDInsight | (Security Baseline)
- Azure Databricks | (Security Baseline)

Workload

- Data analytics
- Enterprise BI

Data type

- Historical data from multiple sources.
- Usually denormalized in a "star" or "snowflake" schema, consisting of fact and dimension tables.
- Usually loaded with new data on a scheduled basis.
- Dimension tables often include multiple historic versions of an entity, referred to as a *slowly changing dimension*.

Examples

- Enterprise data warehouse

Column-family databases

A column-family database organizes data into rows and columns. In its simplest form, a column-family database can appear very similar to a relational database, at least conceptually. The real power of a column-family

database lies in its denormalized approach to structuring sparse data.

You can think of a column-family database as holding tabular data with rows and columns, but the columns are divided into groups known as *column families*. Each column family holds a set of columns that are logically related together and are typically retrieved or manipulated as a unit. Other data that is accessed separately can be stored in separate column families. Within a column family, new columns can be added dynamically, and rows can be sparse (that is, a row doesn't need to have a value for every column).

The following diagram shows an example with two column families, `Identity` and `Contact Info`. The data for a single entity has the same row key in each column-family. This structure, where the rows for any given object in a column family can vary dynamically, is an important benefit of the column-family approach, making this form of data store highly suited for storing structured, volatile data.

CustomerID	Column Family: Identity
001	**First name:** Mu Bae **Last name:** Min
002	**First name:** Francisco **Last name:** Vila Nova **Suffix:** Jr.
003	**First name:** Lena **Last name:** Adamcyz **Title:** Dr.

CustomerID	Column Family: Contact Info
001	**Phone number:** 555-0100 **Email:** someone@example.com
002	**Email:** vilanova@contoso.com
003	**Phone number:** 555-0120

Unlike a key/value store or a document database, most column-family databases store data in key order, rather than by computing a hash. Many implementations allow you to create indexes over specific columns in a column-family. Indexes let you retrieve data by columns value, rather than row key.

Read and write operations for a row are usually atomic with a single column-family, although some implementations provide atomicity across the entire row, spanning multiple column-families.

Azure services

- Azure Cosmos DB Cassandra API | (Security Baseline)
- HBase in HDInsight | (Security Baseline)

Workload

- Most column-family databases perform write operations extremely quickly.
- Update and delete operations are rare.
- Designed to provide high throughput and low-latency access.
- Supports easy query access to a particular set of fields within a much larger record.
- Massively scalable.

Data type

- Data is stored in tables consisting of a key column and one or more column families.
- Specific columns can vary by individual rows.
- Individual cells are accessed via get and put commands
- Multiple rows are returned using a scan command.

Examples

- Recommendations
- Personalization
- Sensor data
- Telemetry
- Messaging
- Social media analytics

- Web analytics
- Activity monitoring
- Weather and other time-series data

Search Engine Databases

A search engine database allows applications to search for information held in external data stores. A search engine database can index massive volumes of data and provide near real-time access to these indexes.

Indexes can be multi-dimensional and may support free-text searches across large volumes of text data. Indexing can be performed using a pull model, triggered by the search engine database, or using a push model, initiated by external application code.

Searching can be exact or fuzzy. A fuzzy search finds documents that match a set of terms and calculates how closely they match. Some search engines also support linguistic analysis that can return matches based on synonyms, genre expansions (for example, matching `dogs` to `pets`), and stemming (matching words with the same root).

Azure service

- Azure Search | (Security Baseline)

Workload

- Data indexes from multiple sources and services.
- Queries are ad-hoc and can be complex.
- Full text search is required.
- Ad hoc self-service query is required.

Data type

- Semi-structured or unstructured text
- Text with reference to structured data

Examples

- Product catalogs
- Site search
- Logging

Time series databases

Time series data is a set of values organized by time. Time series databases typically collect large amounts of data in real time from a large number of sources. Updates are rare, and deletes are often done as bulk operations. Although the records written to a time-series database are generally small, there are often a large number of records, and total data size can grow rapidly.

Azure service

- Azure Time Series Insights

Workload

- Records are generally appended sequentially in time order.
- An overwhelming proportion of operations (95-99%) are writes.
- Updates are rare.
- Deletes occur in bulk, and are made to contiguous blocks or records.
- Data is read sequentially in either ascending or descending time order, often in parallel.

Data type

- A timestamp is used as the primary key and sorting mechanism.
- Tags may define additional information about the type, origin, and other information about the entry.

Examples

- Monitoring and event telemetry.
- Sensor or other IoT data.

Object storage

Object storage is optimized for storing and retrieving large binary objects (images, files, video and audio streams, large application data objects and documents, virtual machine disk images). Large data files are also popularly used in this model, for example, delimiter file (CSV), parquet, and ORC. Object stores can manage extremely large amounts of unstructured data.

Azure service

- Azure Blob Storage | (Security Baseline)
- Azure Data Lake Storage Gen2 | (Security Baseline)

Workload

- Identified by key.
- Content is typically an asset such as a delimiter, image, or video file.
- Content must be durable and external to any application tier.

Data type

- Data size is large.
- Value is opaque.

Examples

- Images, videos, office documents, PDFs
- Static HTML, JSON, CSS
- Log and audit files
- Database backups

Shared files

Sometimes, using simple flat files can be the most effective means of storing and retrieving information. Using file shares enables files to be accessed across a network. Given appropriate security and concurrent access control mechanisms, sharing data in this way can enable distributed services to provide highly scalable data access for performing basic, low-level operations such as simple read and write requests.

Azure service

- Azure Files | (Security Baseline)

Workload

- Migration from existing apps that interact with the file system.
- Requires SMB interface.

Data type

- Files in a hierarchical set of folders.
- Accessible with standard I/O libraries.

Examples

- Legacy files

- Shared content accessible among a number of VMs or app instances

Aided with this understanding of different data storage models, the next step is to evaluate your workload and application, and decide which data store will meet your specific needs. Use the data storage decision tree to help with this process.

Select an Azure data store for your application

11/19/2021 • 2 minutes to read • Edit Online

Azure offers a number of managed data storage solutions, each providing different features and capabilities. This article will help you to choose a managed data store for your application.

If your application consists of multiple workloads, evaluate each workload separately. A complete solution may incorporate multiple data stores.

Select a candidate

Use the following flowchart to select a candidate Azure managed data store.

The output from this flowchart is a **starting point** for consideration. Next, perform a more detailed evaluation of the data store to see if it meets your needs. Refer to Criteria for choosing a data store to aid in this evaluation.

Choose specialized storage

Alternative database solutions often require specific storage solutions. For example, SAP HANA on VMs often employs Azure NetApp Files as its underlying storage solution. Evaluate your vendor's requirements to find an appropriate storage solution to meet your database's requirements. For more information about selecting a storage solution, see Review your storage options.

Criteria for choosing a data store

11/19/2021 • 3 minutes to read • Edit Online

This article describes the comparison criteria you should use when evaluating a data store. The goal is to help you determine which data storage types can meet your solution's requirements.

General considerations

Keep the following considerations in mind when making your selection.

Functional requirements

- **Data format**. What type of data are you intending to store? Common types include transactional data, JSON objects, telemetry, search indexes, or flat files.
- **Data size**. How large are the entities you need to store? Will these entities need to be maintained as a single document, or can they be split across multiple documents, tables, collections, and so forth?
- **Scale and structure**. What is the overall amount of storage capacity you need? Do you anticipate partitioning your data?
- **Data relationships**. Will your data need to support one-to-many or many-to-many relationships? Are relationships themselves an important part of the data? Will you need to join or otherwise combine data from within the same dataset, or from external datasets?
- **Consistency model**. How important is it for updates made in one node to appear in other nodes, before further changes can be made? Can you accept eventual consistency? Do you need ACID guarantees for transactions?
- **Schema flexibility**. What kind of schemas will you apply to your data? Will you use a fixed schema, a schema-on-write approach, or a schema-on-read approach?
- **Concurrency**. What kind of concurrency mechanism do you want to use when updating and synchronizing data? Will the application perform many updates that could potentially conflict. If so, you may require record locking and pessimistic concurrency control. Alternatively, can you support optimistic concurrency controls? If so, is simple timestamp-based concurrency control enough, or do you need the added functionality of multi-version concurrency control?
- **Data movement**. Will your solution need to perform ETL tasks to move data to other stores or data warehouses?
- **Data lifecycle**. Is the data write-once, read-many? Can it be moved into cool or cold storage?
- **Other supported features**. Do you need any other specific features, such as schema validation, aggregation, indexing, full-text search, MapReduce, or other query capabilities?

Non-functional requirements

- **Performance and scalability**. What are your data performance requirements? Do you have specific requirements for data ingestion rates and data processing rates? What are the acceptable response times for querying and aggregation of data once ingested? How large will you need the data store to scale up? Is your workload more read-heavy or write-heavy?
- **Reliability**. What overall SLA do you need to support? What level of fault-tolerance do you need to provide for data consumers? What kind of backup and restore capabilities do you need?

- **Replication**. Will your data need to be distributed among multiple replicas or regions? What kind of data replication capabilities do you require?
- **Limits**. Will the limits of a particular data store support your requirements for scale, number of connections, and throughput?

Management and cost

- **Managed service**. When possible, use a managed data service, unless you require specific capabilities that can only be found in an IaaS-hosted data store.
- **Region availability**. For managed services, is the service available in all Azure regions? Does your solution need to be hosted in certain Azure regions?
- **Portability**. Will your data need to be migrated to on-premises, external datacenters, or other cloud hosting environments?
- **Licensing**. Do you have a preference of a proprietary versus OSS license type? Are there any other external restrictions on what type of license you can use?
- **Overall cost**. What is the overall cost of using the service within your solution? How many instances will need to run, to support your uptime and throughput requirements? Consider operations costs in this calculation. One reason to prefer managed services is the reduced operational cost.
- **Cost effectiveness**. Can you partition your data, to store it more cost effectively? For example, can you move large objects out of an expensive relational database into an object store?

Security

- **Security**. What type of encryption do you require? Do you need encryption at rest? What authentication mechanism do you want to use to connect to your data?
- **Auditing**. What kind of audit log do you need to generate?
- **Networking requirements**. Do you need to restrict or otherwise manage access to your data from other network resources? Does data need to be accessible only from inside the Azure environment? Does the data need to be accessible from specific IP addresses or subnets? Does it need to be accessible from applications or services hosted on-premises or in other external datacenters?

DevOps

- **Skill set**. Are there particular programming languages, operating systems, or other technology that your team is particularly adept at using? Are there others that would be difficult for your team to work with?
- **Clients** Is there good client support for your development languages?

Choose a data pipeline orchestration technology in Azure

11/19/2021 • 2 minutes to read • Edit Online

Most big data solutions consist of repeated data processing operations, encapsulated in workflows. A pipeline orchestrator is a tool that helps to automate these workflows. An orchestrator can schedule jobs, execute workflows, and coordinate dependencies among tasks.

What are your options for data pipeline orchestration?

In Azure, the following services and tools will meet the core requirements for pipeline orchestration, control flow, and data movement:

- Azure Data Factory
- Oozie on HDInsight
- SQL Server Integration Services (SSIS)

These services and tools can be used independently from one another, or used together to create a hybrid solution. For example, the Integration Runtime (IR) in Azure Data Factory V2 can natively execute SSIS packages in a managed Azure compute environment. While there is some overlap in functionality between these services, there are a few key differences.

Key Selection Criteria

To narrow the choices, start by answering these questions:

- Do you need big data capabilities for moving and transforming your data? Usually this means multi-gigabytes to terabytes of data. If yes, then narrow your options to those that best suited for big data.
- Do you require a managed service that can operate at scale? If yes, select one of the cloud-based services that aren't limited by your local processing power.
- Are some of your data sources located on-premises? If yes, look for options that can work with both cloud and on-premises data sources or destinations.
- Is your source data stored in Blob storage on an HDFS filesystem? If so, choose an option that supports Hive queries.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE DATA FACTORY	SQL SERVER INTEGRATION SERVICES (SSIS)	OOZIE ON HDINSIGHT
Managed	Yes	No	Yes
Cloud-based	Yes	No (local)	Yes

CAPABILITY	AZURE DATA FACTORY	SQL SERVER INTEGRATION SERVICES (SSIS)	OOZIE ON HDINSIGHT
Prerequisite	Azure Subscription	SQL Server	Azure Subscription, HDInsight cluster
Management tools	Azure Portal, PowerShell, CLI, .NET SDK	SSMS, PowerShell	Bash shell, Oozie REST API, Oozie web UI
Pricing	Pay per usage	Licensing / pay for features	No additional charge on top of running the HDInsight cluster

Pipeline capabilities

CAPABILITY	AZURE DATA FACTORY	SQL SERVER INTEGRATION SERVICES (SSIS)	OOZIE ON HDINSIGHT
Copy data	Yes	Yes	Yes
Custom transformations	Yes	Yes	Yes (MapReduce, Pig, and Hive jobs)
Azure Machine Learning scoring	Yes	Yes (with scripting)	No
HDInsight On-Demand	Yes	No	No
Azure Batch	Yes	No	No
Pig, Hive, MapReduce	Yes	No	Yes
Spark	Yes	No	No
Execute SSIS Package	Yes	Yes	No
Control flow	Yes	Yes	Yes
Access on-premises data	Yes	Yes	No

Scalability capabilities

CAPABILITY	AZURE DATA FACTORY	SQL SERVER INTEGRATION SERVICES (SSIS)	OOZIE ON HDINSIGHT
Scale up	Yes	No	No
Scale out	Yes	No	Yes (by adding worker nodes to cluster)
Optimized for big data	Yes	No	Yes

Choose a real-time message ingestion technology in Azure

11/19/2021 • 2 minutes to read • Edit Online

Real time processing deals with streams of data that are captured in real-time and processed with minimal latency. Many real-time processing solutions need a message ingestion store to act as a buffer for messages, and to support scale-out processing, reliable delivery, and other message queuing semantics.

What are your options for real-time message ingestion?

- Azure Event Hubs
- Azure IoT Hub
- Kafka on HDInsight

Azure Event Hubs

Azure Event Hubs is a highly scalable data streaming platform and event ingestion service, capable of receiving and processing millions of events per second. Event Hubs can process and store events, data, or telemetry produced by distributed software and devices. Data sent to an event hub can be transformed and stored using any real-time analytics provider or batching/storage adapters. Event Hubs provides publish-subscribe capabilities with low latency at massive scale, which makes it appropriate for big data scenarios.

Azure IoT Hub

Azure IoT Hub is a managed service that enables reliable and secure bidirectional communications between millions of IoT devices and a cloud-based back end.

Features of IoT Hub include:

- Multiple options for device-to-cloud and cloud-to-device communication. These options include one-way messaging, file transfer, and request-reply methods.
- Message routing to other Azure services.
- Queryable store for device metadata and synchronized state information.
- Secure communications and access control using per-device security keys or X.509 certificates.
- Monitoring of device connectivity and device identity management events.

In terms of message ingestion, IoT Hub is similar to Event Hubs. However, it was specifically designed for managing IoT device connectivity, not just message ingestion. For more information, see Comparison of Azure IoT Hub and Azure Event Hubs.

Kafka on HDInsight

Apache Kafka is an open-source distributed streaming platform that can be used to build real-time data pipelines and streaming applications. Kafka also provides message broker functionality similar to a message queue, where you can publish and subscribe to named data streams. It is horizontally scalable, fault-tolerant, and extremely fast. Kafka on HDInsight provides a Kafka as a managed, highly scalable, and highly available service in Azure.

Some common use cases for Kafka are:

- **Messaging**. Because it supports the publish-subscribe message pattern, Kafka is often used as a message broker.
- **Activity tracking**. Because Kafka provides in-order logging of records, it can be used to track and re-create activities, such as user actions on a web site.
- **Aggregation**. Using stream processing, you can aggregate information from different streams to combine and centralize the information into operational data.
- **Transformation**. Using stream processing, you can combine and enrich data from multiple input topics into one or more output topics.

Key selection criteria

To narrow the choices, start by answering these questions:

- Do you need two-way communication between your IoT devices and Azure? If so, choose IoT Hub.
- Do you need to manage access for individual devices and be able to revoke access to a specific device? If yes, choose IoT Hub.

Capability matrix

The following tables summarize the key differences in capabilities.

CAPABILITY	IOT HUB	EVENT HUBS	KAFKA ON HDINSIGHT
Cloud-to-device communications	Yes	No	No
Device-initiated file upload	Yes	No	No
Device state information	Device twins	No	No
Protocol support	MQTT, AMQP, HTTPS [1]	AMQP, HTTPS, Kafka Protocol	Kafka Protocol
Security	Per-device identity; revocable access control.	Shared access policies; limited revocation through publisher policies.	Authentication using SASL; pluggable authorization; integration with external authentication services supported.

[1] You can also use Azure IoT protocol gateway as a custom gateway to enable protocol adaptation for IoT Hub.

For more information, see Comparison of Azure IoT Hub and Azure Event Hubs.

Choose a search data store in Azure

11/19/2021 • 2 minutes to read • Edit Online

This article compares technology choices for search data stores in Azure. A search data store is used to create and store specialized indexes for performing searches on free-form text. The text that is indexed may reside in a separate data store, such as blob storage. An application submits a query to the search data store, and the result is a list of matching documents. For more information about this scenario, see Processing free-form text for search.

What are your options when choosing a search data store?

In Azure, all of the following data stores will meet the core requirements for search against free-form text data by providing a search index:

- Azure Cognitive Search
- Elasticsearch
- HDInsight with Solr
- Azure SQL Database with full text search

Key selection criteria

For search scenarios, begin choosing the appropriate search data store for your needs by answering these questions:

- Do you want a managed service rather than managing your own servers?
- Can you specify your index schema at design time? If not, choose an option that supports updateable schemas.
- Do you need an index only for full-text search, or do you also need rapid aggregation of numeric data and other analytics? If you need functionality beyond full-text search, consider options that support additional analytics.
- Do you need a search index for log analytics, with support for log collection, aggregation, and visualizations on indexed data? If so, consider Elasticsearch, which is part of a log analytics stack.
- Do you need to index data in common document formats such as PDF, Word, PowerPoint, and Excel? If yes, choose an option that provides document indexers.
- Does your database have specific security needs? If yes, consider the security features listed below.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	COGNITIVE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Is managed service	Yes	No	Yes	Yes

CAPABILITY	COGNITIVE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
REST API	Yes	Yes	Yes	No
Programmability	.NET, Java, Python, JavaScript	Java	Java	T-SQL
Document indexers for common file types (PDF, DOCX, TXT, and so on)	Yes	No	Yes	No

Manageability capabilities

CAPABILITY	COGNITIVE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Updateable schema	Yes	Yes	Yes	Yes
Supports scale out	Yes	Yes	Yes	No

Analytic workload capabilities

CAPABILITY	COGNITIVE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Supports analytics beyond full text search	No	Yes	Yes	Yes
Part of a log analytics stack	No	Yes (ELK)	No	No
Supports semantic search	Yes (find similar documents only)	Yes	Yes	Yes

Security capabilities

CAPABILITY	COGNITIVE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Row-level security	Partial (requires application query to filter by group id)	Partial (requires application query to filter by group id)	Yes	Yes
Transparent data encryption	No	No	No	Yes
Restrict access to specific IP addresses	Yes	Yes	Yes	Yes
Restrict access to allow virtual network access only	Yes	Yes	Yes	Yes

CAPABILITY	COGNITIVE SEARCH	ELASTICSEARCH	HDINSIGHT WITH SOLR	SQL DATABASE
Active Directory authentication (integrated authentication)	No	No	No	Yes

See also

Processing free-form text for search

Choose a stream processing technology in Azure

11/19/2021 • 2 minutes to read • Edit Online

This article compares technology choices for real-time stream processing in Azure.

Real-time stream processing consumes messages from either queue or file-based storage, processes the messages, and forwards the result to another message queue, file store, or database. Processing may include querying, filtering, and aggregating messages. Stream processing engines must be able to consume endless streams of data and produce results with minimal latency. For more information, see Real time processing.

What are your options when choosing a technology for real-time processing?

In Azure, all of the following data stores will meet the core requirements supporting real-time processing:

- Azure Stream Analytics
- HDInsight with Spark Streaming
- Apache Spark in Azure Databricks
- HDInsight with Storm
- Azure Functions
- Azure App Service WebJobs
- Apache Kafka streams API

Key Selection Criteria

For real-time processing scenarios, begin choosing the appropriate service for your needs by answering these questions:

- Do you prefer a declarative or imperative approach to authoring stream processing logic?
- Do you need built-in support for temporal processing or windowing?
- Does your data arrive in formats besides Avro, JSON, or CSV? If yes, consider options that support any format using custom code.
- Do you need to scale your processing beyond 1 GB/s? If yes, consider the options that scale with the cluster size.

Capability matrix

The following tables summarize the key differences in capabilities.

General capabilities

CAPABILITY	AZURE STREAM ANALYTICS	HDINSIGHT WITH SPARK STREAMING	APACHE SPARK IN AZURE DATABRICKS	HDINSIGHT WITH STORM	AZURE FUNCTIONS	AZURE APP SERVICE WEBJOBS

CAPABILITY	AZURE STREAM ANALYTICS	HDINSIGHT WITH SPARK STREAMING	APACHE SPARK IN AZURE DATABRICKS	HDINSIGHT WITH STORM	AZURE FUNCTIONS	AZURE APP SERVICE WEBJOBS
Programmability	Stream analytics query language, JavaScript	C#/F#, Java, Python, Scala	C#/F#, Java, Python, R, Scala	C#, Java	C#, F#, Java, Node.js, Python	C#, Java, Node.js, PHP, Python
Programming paradigm	Declarative	Mixture of declarative and imperative	Mixture of declarative and imperative	Imperative	Imperative	Imperative
Pricing model	Streaming units	Per cluster hour	Databricks units	Per cluster hour	Per function execution and resource consumption	Per app service plan hour

Integration capabilities

CAPABILITY	AZURE STREAM ANALYTICS	HDINSIGHT WITH SPARK STREAMING	APACHE SPARK IN AZURE DATABRICKS	HDINSIGHT WITH STORM	AZURE FUNCTIONS	AZURE APP SERVICE WEBJOBS
Inputs	Azure Event Hubs, Azure IoT Hub, Azure Blob storage	Event Hubs, IoT Hub, Kafka, HDFS, Storage Blobs, Azure Data Lake Store	Event Hubs, IoT Hub, Kafka, HDFS, Storage Blobs, Azure Data Lake Store	Event Hubs, IoT Hub, Storage Blobs, Azure Data Lake Store	Supported bindings	Service Bus, Storage Queues, Storage Blobs, Event Hubs, WebHooks, Cosmos DB, Files
Sinks	Azure Data Lake Store, Azure SQL Database, Storage Blobs, Event Hubs, Power BI, Table Storage, Service Bus Queues, Service Bus Topics, Cosmos DB, Azure Functions	HDFS, Kafka, Storage Blobs, Azure Data Lake Store, Cosmos DB	HDFS, Kafka, Storage Blobs, Azure Data Lake Store, Cosmos DB	Event Hubs, Service Bus, Kafka	Supported bindings	Service Bus, Storage Queues, Storage Blobs, Event Hubs, WebHooks, Cosmos DB, Files

Processing capabilities

CAPABILITY	AZURE STREAM ANALYTICS	HDINSIGHT WITH SPARK STREAMING	APACHE SPARK IN AZURE DATABRICKS	HDINSIGHT WITH STORM	AZURE FUNCTIONS	AZURE APP SERVICE WEBJOBS
Built-in temporal/windowing support	Yes	Yes	Yes	Yes	No	No
Input data formats	Avro, JSON or CSV, UTF-8 encoded	Any format using custom code	Any format using custom code	Any format using custom code	Any format using custom code	Any format using custom code
Scalability	Query partitions	Bounded by cluster size	Bounded by Databricks cluster scale configuration	Bounded by cluster size	Up to 200 function app instances processing in parallel	Bounded by app service plan capacity
Late arrival and out of order event handling support	Yes	Yes	Yes	Yes	No	No

See also:

- Choosing a real-time message ingestion technology
- Real time processing

Data management patterns

11/19/2021 • 2 minutes to read • Edit Online

Data management is the key element of cloud applications, and influences most of the quality attributes. Data is typically hosted in different locations and across multiple servers for reasons such as performance, scalability or availability, and this can present a range of challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations.

Additionally data should be protected at rest, in transit, and via authorized access mechanisms to maintain security assurances of confidentiality, integrity, and availability. Refer to the Azure Security Benchmark Data Protection Control for more information.

PATTERN	SUMMARY
Cache-Aside	Load data on demand into a cache from a data store
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Sharding	Divide a data store into a set of horizontal partitions or shards.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

Transfer data to and from Azure

11/19/2021 • 7 minutes to read • Edit Online

There are several options for transferring data to and from Azure, depending on your needs.

Physical transfer

Using physical hardware to transfer data to Azure is a good option when:

- Your network is slow or unreliable.
- Getting additional network bandwidth is cost-prohibitive.
- Security or organizational policies do not allow outbound connections when dealing with sensitive data.

If your primary concern is how long it will take to transfer your data, you may want to run a test to verify whether network transfer is actually slower than physical transport.

There are two main options for physically transporting data to Azure:

- **Azure Import/Export**. The Azure Import/Export service lets you securely transfer large amounts of data to Azure Blob Storage or Azure Files by shipping internal SATA HDDs or SDDs to an Azure datacenter. You can also use this service to transfer data from Azure Storage to hard disk drives and have these shipped to you for loading on-premises.

- **Azure Data Box**. Azure Data Box is a Microsoft-provided appliance that works much like the Azure Import/Export service. Microsoft ships you a proprietary, secure, and tamper-resistant transfer appliance and handles the end-to-end logistics, which you can track through the portal. One benefit of the Azure Data Box service is ease of use. You don't need to purchase several hard drives, prepare them, and transfer files to each one. Azure Data Box is supported by a number of industry-leading Azure partners to make it easier to seamlessly use offline transport to the cloud from their products.

Command line tools and APIs

Consider these options when you want scripted and programmatic data transfer.

- **Azure CLI**. The Azure CLI is a cross-platform tool that allows you to manage Azure services and upload data to Azure Storage.

- **AzCopy**. Use AzCopy from a Windows or Linux command-line to easily copy data to and from Azure Blob, File, and Table storage with optimal performance. AzCopy supports concurrency and parallelism, and the ability to resume copy operations when interrupted. You can also use AzCopy to copy data from AWS to Azure. For programmatic access, the Microsoft Azure Storage Data Movement Library is the core framework that powers AzCopy. It is provided as a .NET Core library.

- **PowerShell**. The `Start-AzureStorageBlobCopy` PowerShell cmdlet is an option for Windows administrators who are used to PowerShell.

- **AdlCopy**. AdlCopy enables you to copy data from Azure Storage Blobs into Data Lake Store. It can also be used to copy data between two Azure Data Lake Store accounts. However, it cannot be used to copy data from Data Lake Store to Storage Blobs.

- **Distcp**. If you have an HDInsight cluster with access to Data Lake Store, you can use Hadoop ecosystem tools like Distcp to copy data to and from an HDInsight cluster storage (WASB) into a Data Lake Store account.

- **Sqoop**. Sqoop is an Apache project and part of the Hadoop ecosystem. It comes preinstalled on all HDInsight clusters. It allows data transfer between an HDInsight cluster and relational databases such as SQL, Oracle, MySQL, and so on. Sqoop is a collection of related tools, including import and export. Sqoop works with HDInsight clusters using either Azure Storage blobs or Data Lake Store attached storage.
- **PolyBase**. PolyBase is a technology that accesses data outside of the database through the T-SQL language. In SQL Server 2016, it allows you to run queries on external data in Hadoop or to import/export data from Azure Blob Storage. In Azure Synapse Analytics, you can import/export data from Azure Blob Storage and Azure Data Lake Store. Currently, PolyBase is the fastest method of importing data into Azure Synapse.
- **Hadoop command line**. When you have data that resides on an HDInsight cluster head node, you can use the `hadoop -copyFromLocal` command to copy that data to your cluster's attached storage, such as Azure Storage blob or Azure Data Lake Store. In order to use the Hadoop command, you must first connect to the head node. Once connected, you can upload a file to storage.

Graphical interface

Consider the following options if you are only transferring a few files or data objects and don't need to automate the process.

- **Azure Storage Explorer**. Azure Storage Explorer is a cross-platform tool that lets you manage the contents of your Azure storage accounts. It allows you to upload, download, and manage blobs, files, queues, tables, and Azure Cosmos DB entities. Use it with Blob storage to manage blobs and folders, as well as upload and download blobs between your local file system and Blob storage, or between storage accounts.
- **Azure portal**. Both Blob storage and Data Lake Store provide a web-based interface for exploring files and uploading new files one at a time. This is a good option if you do not want to install any tools or issue commands to quickly explore your files, or to simply upload a handful of new ones.

Data pipeline

Azure Data Factory. Azure Data Factory is a managed service best suited for regularly transferring files between a number of Azure services, on-premises, or a combination of the two. Using Azure Data Factory, you can create and schedule data-driven workflows (called pipelines) that ingest data from disparate data stores. It can process and transform the data by using compute services such as Azure HDInsight Hadoop, Spark, Azure Data Lake Analytics, and Azure Machine Learning. Create data-driven workflows for orchestrating and automating data movement and data transformation.

Key Selection Criteria

For data transfer scenarios, choose the appropriate system for your needs by answering these questions:

- Do you need to transfer very large amounts of data, where doing so over an Internet connection would take too long, be unreliable, or too expensive? If yes, consider physical transfer.
- Do you prefer to script your data transfer tasks, so they are reusable? If so, select one of the command line options or Azure Data Factory.
- Do you need to transfer a very large amount of data over a network connection? If so, select an option that is optimized for big data.
- Do you need to transfer data to or from a relational database? If yes, choose an option that supports one or more relational databases. Note that some of these options also require a Hadoop cluster.

- Do you need an automated data pipeline or workflow orchestration? If yes, consider Azure Data Factory.

Capability matrix

The following tables summarize the key differences in capabilities.

Physical transfer

CAPABILITY	AZURE IMPORT/EXPORT SERVICE	AZURE DATA BOX
Form factor	Internal SATA HDDs or SDDs	Secure, tamper-proof, single hardware appliance
Microsoft manages shipping logistics	No	Yes
Integrates with partner products	No	Yes
Custom appliance	No	Yes

Command line tools

Hadoop/HDInsight:

CAPABILITY	DISTCP	SQOOP	HADOOP CLI
Optimized for big data	Yes	Yes	Yes
Copy to relational database	No	Yes	No
Copy from relational database	No	Yes	No
Copy to Blob storage	Yes	Yes	Yes
Copy from Blob storage	Yes	Yes	No
Copy to Data Lake Store	Yes	Yes	Yes
Copy from Data Lake Store	Yes	Yes	No

Other:

CAPABILITY	AZURE CLI	AZCOPY	POWERSHELL	ADLCOPY	POLYBASE
Compatible platforms	Linux, OS X, Windows	Linux, Windows	Windows	Linux, OS X, Windows	SQL Server, Azure Synapse
Optimized for big data	No	Yes	No	Yes [1]	Yes [2]
Copy to relational database	No	No	No	No	Yes

CAPABILITY	AZURE CLI	AZCOPY	POWERSHELL	ADLCOPY	POLYBASE
Copy from relational database	No	No	No	No	Yes
Copy to Blob storage	Yes	Yes	Yes	No	Yes
Copy from Blob storage	Yes	Yes	Yes	Yes	Yes
Copy to Data Lake Store	No	Yes	Yes	Yes	Yes
Copy from Data Lake Store	No	No	Yes	Yes	Yes

[1] AdlCopy is optimized for transferring big data when used with a Data Lake Analytics account.

[2] PolyBase performance can be increased by pushing computation to Hadoop and using PolyBase scale-out groups to enable parallel data transfer between SQL Server instances and Hadoop nodes.

Graphical interface and Azure Data Factory

CAPABILITY	AZURE STORAGE EXPLORER	AZURE PORTAL *	AZURE DATA FACTORY
Optimized for big data	No	No	Yes
Copy to relational database	No	No	Yes
Copy from relational database	No	No	Yes
Copy to Blob storage	Yes	No	Yes
Copy from Blob storage	Yes	No	Yes
Copy to Data Lake Store	No	No	Yes
Copy from Data Lake Store	No	No	Yes
Upload to Blob storage	Yes	Yes	Yes
Upload to Data Lake Store	Yes	Yes	Yes
Orchestrate data transfers	No	No	Yes
Custom data transformations	No	No	Yes
Pricing model	Free	Free	Pay per usage

* Azure portal in this case means using the web-based exploration tools for Blob storage and Data Lake Store.

Extend on-premises data solutions to the cloud

11/19/2021 • 7 minutes to read • Edit Online

When organizations move workloads and data to the cloud, their on-premises datacenters often continue to play an important role. The term *hybrid cloud* refers to a combination of public cloud and on-premises datacenters, to create an integrated IT environment that spans both. Some organizations use hybrid cloud as a path to migrate their entire datacenter to the cloud over time. Other organizations use cloud services to extend their existing on-premises infrastructure.

This article describes some considerations and best practices for managing data in a hybrid cloud solution,

When to use a hybrid solution

Consider using a hybrid solution in the following scenarios:

- As a transition strategy during a longer-term migration to a fully cloud-native solution.
- When regulations or policies do not permit moving specific data or workloads to the cloud.
- For disaster recovery and fault tolerance, by replicating data and services between on-premises and cloud environments.
- To reduce latency between your on-premises datacenter and remote locations, by hosting part of your architecture in Azure.

Challenges

- Creating a consistent environment in terms of security, management, and development, and avoiding duplication of work.
- Creating a reliable, low latency and secure data connection between your on-premises and cloud environments.
- Replicating your data and modifying applications and tools to use the correct data stores within each environment.
- Securing and encrypting data that is hosted in the cloud but accessed from on-premises, or vice versa.

On-premises data stores

On-premises data stores include databases and files. There may be several reasons to keep these data stores local. There may be regulations or policies that do not permit moving specific data or workloads to the cloud. Data sovereignty, privacy, or security concerns may favor on-premises placement. During a migration, you may want to keep some data local to an application that hasn't been migrated yet.

Considerations in placing application data in a public cloud include:

- **Cost**. The cost of storage in Azure can be significantly lower than the cost of maintaining storage with similar characteristics in an on-premises datacenter. Of course, many companies have existing investments in high-end SANs, so these cost advantages may not reach full fruition until existing hardware ages out.
- **Elastic scale**. Planning and managing data capacity growth in an on-premises environment can be challenging, particularly when data growth is difficult to predict. These applications can take advantage of the capacity-on-demand and virtually unlimited storage available in the cloud. This consideration is less

relevant for applications that consist of relatively static-sized datasets.

- **Disaster recovery**. Data stored in Azure can be automatically replicated within an Azure region and across geographic regions. In hybrid environments, these same technologies can be used to replicate between on-premises and cloud-based data stores.

Extending data stores to the cloud

There are several options for extending on-premises data stores to the cloud. One option is to have on-premises and cloud replicas. This can help achieve a high level of fault tolerance, but may require making changes to applications to connect to the appropriate data store in the event of a failover.

Another option is to move a portion of the data to cloud storage, while keeping the more current or more highly accessed data on-premises. This method can provide a more cost-effective option for long-term storage, as well as improve data access response times by reducing your operational data set.

A third option is to keep all data on-premises, but use cloud computing to host applications. To do this, you would host your application in the cloud and connect it to your on-premises data store over a secure connection.

Azure Stack

For a complete hybrid cloud solution, consider using Microsoft Azure Stack. Azure Stack is a hybrid cloud platform that lets you provide Azure services from your datacenter. This helps maintain consistency between on-premises and Azure, by using identical tools and requiring no code changes.

The following are some use cases for Azure and Azure Stack:

- **Edge and disconnected solutions**. Address latency and connectivity requirements by processing data locally in Azure Stack and then aggregating in Azure for further analytics, with common application logic across both.
- **Cloud applications that meet varied regulations**. Develop and deploy applications in Azure, with the flexibility to deploy the same applications on-premises on Azure Stack to meet regulatory or policy requirements.
- **Cloud application model on-premises**. Use Azure to update and extend existing applications or build new ones. Use consistent DevOps processes across Azure in the cloud and Azure Stack on-premises.

SQL Server data stores

If you are running SQL Server on-premises, you can use Microsoft Azure Blob Storage service for backup and restore. For more information, see SQL Server Backup and Restore with Microsoft Azure Blob Storage Service. This capability gives you limitless offsite storage, and the ability to share the same backups between SQL Server running on-premises and SQL Server running in a virtual machine in Azure.

Azure SQL Database is a managed relational database-as-a service. Because Azure SQL Database uses the Microsoft SQL Server Engine, applications can access data in the same way with both technologies. Azure SQL Database can also be combined with SQL Server in useful ways. For example, the SQL Server Stretch Database feature lets an application access what looks like a single table in a SQL Server database while some or all rows of that table might be stored in Azure SQL Database. This technology automatically moves data that's not accessed for a defined period of time to the cloud. Applications reading this data are unaware that any data has been moved to the cloud.

Maintaining data stores on-premises and in the cloud can be challenging when you desire to keep the data synchronized. You can address this with SQL Data Sync, a service built on Azure SQL Database that lets you synchronize the data you select, bi-directionally across multiple Azure SQL databases and SQL Server instances.

While Data Sync makes it easy to keep your data up-to-date across these various data stores, it should not be used for disaster recovery or for migrating from on-premises SQL Server to Azure SQL Database.

For disaster recovery and business continuity, you can use AlwaysOn Availability Groups to replicate data across two or more instances of SQL Server, some of which can be running on Azure virtual machines in another geographic region.

Network shares and file-based data stores

In a hybrid cloud architecture, it is common for an organization to keep newer files on-premises while archiving older files to the cloud. This is sometimes called file tiering, where there is seamless access to both sets of files, on-premises and cloud-hosted. This approach helps to minimize network bandwidth usage and access times for newer files, which are likely to be accessed the most often. At the same time, you get the benefits of cloud-based storage for archived data.

Organizations may also wish to move their network shares entirely to the cloud. This would be desirable, for example, if the applications that access them are also located in the cloud. This procedure can be done using data orchestration tools.

Azure StorSimple offers the most complete integrated storage solution for managing storage tasks between your on-premises devices and Azure cloud storage. StorSimple is an efficient, cost-effective, and easily manageable storage area network (SAN) solution that eliminates many of the issues and expenses associated with enterprise storage and data protection. It uses the proprietary StorSimple 8000 series device, integrates with cloud services, and provides a set of integrated management tools.

Another way to use on-premises network shares alongside cloud-based file storage is with Azure Files. Azure Files offers fully managed file shares that you can access with the standard Server Message Block (SMB) protocol (sometimes referred to as CIFS). You can mount Azure Files as a file share on your local computer, or use them with existing applications that access local or network share files.

To synchronize file shares in Azure Files with your on-premises Windows Servers, use Azure File Sync. One major benefit of Azure File Sync is the ability to tier files between your on-premises file server and Azure Files. This lets you keep only the newest and most recently accessed files locally.

For more information, see Deciding when to use Azure Blob storage, Azure Files, or Azure Disks.

Hybrid networking

This article focused on hybrid data solutions, but another consideration is how to extend your on-premises network to Azure. For more information about this aspect of hybrid solutions, see:

- Choose a solution for connecting an on-premises network to Azure
- Hybrid network reference architectures

Securing data solutions

11/19/2021 • 5 minutes to read • Edit Online

For many, making data accessible in the cloud, particularly when transitioning from working exclusively in on-premises data stores, can cause some concern around increased accessibility to that data and new ways in which to secure it.

Challenges

- Centralizing the monitoring and analysis of security events stored in numerous logs.
- Implementing encryption and authorization management across your applications and services.
- Ensuring that centralized identity management works across all of your solution components, whether on-premises or in the cloud.

Data Protection

The first step to protecting information is identifying what to protect. Develop clear, simple, and well-communicated guidelines to identify, protect, and monitor the most important data assets anywhere they reside. Establish the strongest protection for assets that have a disproportionate impact on the organization's mission or profitability. These are known as high value assets, or HVAs. Perform stringent analysis of HVA lifecycle and security dependencies, and establish appropriate security controls and conditions. Similarly, identify and classify sensitive assets, and define the technologies and processes to automatically apply security controls.

Once the data you need to protect has been identified, consider how you will protect the data *at rest* and data *in transit*.

- **Data at rest**: Data that exists statically on physical media, whether magnetic or optical disk, on premises or in the cloud.
- **Data in transit**: Data while it is being transferred between components, locations or programs, such as over the network, across a service bus (from on-premises to cloud and vice-versa), or during an input/output process.

To learn more about protecting your data at rest or in transit, see Azure Data Security and Encryption Best Practices.

Access Control

Central to protecting your data in the cloud is a combination of identity management and access control. Given the variety and type of cloud services, as well as the rising popularity of hybrid cloud, there are several key practices you should follow when it comes to identity and access control:

- Centralize your identity management.
- Enable Single Sign-On (SSO).
- Deploy password management.
- Enforce multi-factor authentication for users.
- Use Azure role-based access control (Azure RBAC).
- Conditional Access Policies should be configured, which enhances the classic concept of user identity with additional properties related to user location, device type, patch level, and so on.
- Control locations where resources are created using resource manager.

- Actively monitor for suspicious activities

For more information, see Azure Identity Management and access control security best practices.

Auditing

Beyond the identity and access monitoring previously mentioned, the services and applications that you use in the cloud should be generating security-related events that you can monitor. The primary challenge to monitoring these events is handling the quantities of logs , in order to avoid potential problems or troubleshoot past ones. Cloud-based applications tend to contain many moving parts, most of which generate some level of logging and telemetry. Use centralized monitoring and analysis to help you manage and make sense of the large amount of information.

For more information, see Azure Logging and Auditing.

Securing data solutions in Azure

Encryption

Virtual machines. Use Azure Disk Encryption to encrypt the attached disks on Windows or Linux VMs. This solution integrates with Azure Key Vault to control and manage the disk-encryption keys and secrets.

Azure Storage. Use Azure Storage Service Encryption to automatically encrypt data at rest in Azure Storage. Encryption, decryption, and key management are totally transparent to users. Data can also be secured in transit by using client-side encryption with Azure Key Vault. For more information, see Client-Side Encryption and Azure Key Vault for Microsoft Azure Storage.

SQL Database and **Azure Synapse Analytics**. Use Transparent Data Encryption (TDE) to perform real-time encryption and decryption of your databases, associated backups, and transaction log files without requiring any changes to your applications. SQL Database can also use Always Encrypted to help protect sensitive data at rest on the server, during movement between client and server, and while the data is in use. You can use Azure Key Vault to store your Always Encrypted encryption keys.

Rights management

Azure Rights Management is a cloud-based service that uses encryption, identity, and authorization policies to secure files and email. It works across multiple devices — phones, tablets, and PCs. Information can be protected both within your organization and outside your organization because that protection remains with the data, even when it leaves your organization's boundaries.

Access control

Use Azure role-based access control (Azure RBAC) to restrict access to Azure resources based on user roles. If you are using Active Directory on-premises, you can synchronize with Azure AD to provide users with a cloud identity based on their on-premises identity.

Use Conditional access in Azure Active Directory to enforce controls on the access to applications in your environment based on specific conditions. For example, your policy statement could take the form of: *When contractors are trying to access our cloud apps from networks that are not trusted, then block access*.

Azure AD Privileged Identity Management can help you manage, control, and monitor your users and what sorts of tasks they are performing with their admin privileges. This is an important step to limiting who in your organization can carry out privileged operations in Azure AD, Azure, Microsoft 365, or SaaS apps, as well as monitor their activities.

Network

To protect data in transit, always use SSL/TLS when exchanging data across different locations. Sometimes you need to isolate your entire communication channel between your on-premises and cloud infrastructure by using

either a virtual private network (VPN) or ExpressRoute. For more information, see Extending on-premises data solutions to the cloud.

Use network security groups to reduce the number of potential attack vectors. A network security group contains a list of security rules that allow or deny inbound or outbound network traffic based on source or destination IP address, port, and protocol.

Use Virtual Network service endpoints to secure Azure SQL or Azure Storage resources, so that only traffic from your virtual network can access these resources.

VMs within an Azure Virtual Network (VNet) can securely communicate with other VNets using virtual network peering. Network traffic between peered virtual networks is private. Traffic between the virtual networks is kept on the Microsoft backbone network.

For more information, see Azure network security

Monitoring

Microsoft Defender for Cloud automatically collects, analyzes, and integrates log data from your Azure resources, the network, and connected partner solutions, such as firewall solutions, to detect real threats and reduce false positives.

Log Analytics provides centralized access to your logs and helps you analyze that data and create custom alerts.

Azure SQL Database Threat Detection detects anomalous activities indicating unusual and potentially harmful attempts to access or exploit databases. Security officers or other designated administrators can receive an immediate notification about suspicious database activities as they occur. Each notification provides details of the suspicious activity and recommends how to further investigate and mitigate the threat.

Tenancy models for SaaS applications

11/19/2021 • 3 minutes to read • Edit Online

In a Software as a Service (SaaS) model, each of your customers are a tenant of your application. Each tenant pays for access to the SaaS application by paying a subscription fee. This article describes the application tenancy models available to SaaS application builders.

When designing a SaaS application, you must choose the application tenancy model that best fits the needs of your customers and your business. In general, the application tenancy model doesn't impact the functionality of an application. But it likely impacts other aspects of the overall solution including scale, tenant isolation, cost per tenant and operation complexity.

What are application tenancy models

The sections below explore the concepts of single, mixed, and multi-tenant application models.

Single tenant

In the single tenant model, a single dedicated instance of an application is deployed for each customer. For example, with a N-tier architecture style application, all customers get a new dedicated instance of the web, middle, and data tiers. These tiers are not shared between these customers.

Mixed tenant

In this model, one or more parts of an application are deployed as dedicated for each customer, and the rest is shared between all customers. For example, with a N-tier architecture style application the web and middle tiers are shared between all customers. However, a dedicated data tier and database is provisioned for each customer.

Multi-tenant

In this model, a single instance of the application is deployed for all customers and shared amongst them. For example, with a N-tier architecture style application, the web, middle and data tiers are shared between all customers.

A combination of these models can be provided for customers with different needs. For example, your basic tier of service would run on a shared multi-tenant instance of your application. As a baseline, your customers can access your app with lower performance or limited functionality for a lower cost. On top of this baseline, a dedicate service tier could run on a single tenant model. For customers that need higher performance or additional functionality, you can provide an isolated instance of your application for a higher cost.

Compare application tenancy models

In general, the tenancy model doesn't impact the functionality of an application, but it likely impacts other aspects of the overall solution. The following table summarizes the differences between the application tenancy models:

MEASUREMENT	SINGLE TENANT	MIXED	MULTI-TENANT
Scale	Medium	High	Very High
Tenant isolation	Very High	High	Low
Cost per tenant	High	Medium	Low
Operational complexity	*Low-High* - Individually simple, complex at scale.	*Low-Medium* - Need to address complexity at scale.	*Low-High* - Individual tenant management is complex.

The measurement terms are as detailed below:

Scale: The number of concurrent customers (or tenants) your application can service.

Tenant isolation: The degree in which customers' data and performance is separated from other customers.

Cost per tenant: The relative amount it costs to run a single customer's tenant.

Operational complexity: How complex it is to achieve operational tasks like automation, security, monitoring, maintenance, metering, application deployments, disaster recovery/BCP.

In general, the single tenancy application model and a small number of tenants, are simpler to manage. With a large number of tenants, it can become highly complex without strong operational excellence practices. Conversely, in the multi-tenancy application model, it is easier to manage the system as a whole, but highly complex when you want to manage an individual customer tenant by itself, without strong operational excellence practices.

Additional considerations

Once you know the customer's needs and business goals, start asking these questions:

- Do I operate in a highly regulated industry that requires customers data to be isolated from other customers?
- Am I looking to rapidly scale my application to many thousands of clients?
- Am I concerned about how much it costs to run each tenant/customer instance?

The answers to these questions will help you refine your tenancy requirement.

Next steps

Read more about application tenancy patterns in Multi-tenant SaaS database tenancy patterns.

Monitoring Azure Databricks

11/19/2021 • 2 minutes to read • Edit Online

Azure Databricks is a fast, powerful Apache Spark–based analytics service that makes it easy to rapidly develop and deploy big data analytics and artificial intelligence (AI) solutions. Many users take advantage of the simplicity of notebooks in their Azure Databricks solutions. For users that require more robust computing options, Azure Databricks supports the distributed execution of custom application code.

Monitoring is a critical part of any production-level solution, and Azure Databricks offers robust functionality for monitoring custom application metrics, streaming query events, and application log messages. Azure Databricks can send this monitoring data to different logging services.

The following articles show how to send monitoring data from Azure Databricks to Azure Monitor, the monitoring data platform for Azure.

- Send Azure Databricks application logs to Azure Monitor
- Use dashboards to visualize Azure Databricks metrics
- Troubleshoot performance bottlenecks

The code library that accompanies these articles extends the core monitoring functionality of Azure Databricks to send Spark metrics, events, and logging information to Azure Monitor.

The audience for these articles and the accompanying code library are Apache Spark and Azure Databricks solution developers. The code must be built into Java Archive (JAR) files and then deployed to an Azure Databricks cluster. The code is a combination of Scala and Java, with a corresponding set of Maven project object model (POM) files to build the output JAR files. Understanding of Java, Scala, and Maven are recommended as prerequisites.

Next steps

Start by building the code library and deploying it to your Azure Databricks cluster.

Send Azure Databricks application logs to Azure Monitor

Send Azure Databricks application logs to Azure Monitor

11/19/2021 • 2 minutes to read • Edit Online

This article shows how to send application logs and metrics from Azure Databricks to a Log Analytics workspace. It uses the Azure Databricks Monitoring Library, which is available on GitHub.

Prerequisites

Configure your Azure Databricks cluster to use the monitoring library, as described in the GitHub readme.

> **NOTE**
>
> The monitoring library streams Apache Spark level events and Spark Structured Streaming metrics from your jobs to Azure Monitor. You don't need to make any changes to your application code for these events and metrics.

Send application metrics using Dropwizard

Spark uses a configurable metrics system based on the Dropwizard Metrics Library. For more information, see Metrics in the Spark documentation.

To send application metrics from Azure Databricks application code to Azure Monitor, follow these steps:

1. Build the **spark-listeners-loganalytics-1.0-SNAPSHOT.jar** JAR file as described in the GitHub readme.

2. Create Dropwizard gauges or counters in your application code. You can use the `UserMetricsSystem` class defined in the monitoring library. The following example creates a counter named `counter1`.

```
import org.apache.spark.metrics.UserMetricsSystems
import org.apache.spark.sql.SparkSession

object StreamingQueryListenerSampleJob  {

  private final val METRICS_NAMESPACE = "samplejob"
  private final val COUNTER_NAME = "counter1"

  def main(args: Array[String]): Unit = {

    val spark = SparkSession
      .builder
      .getOrCreate

    val driverMetricsSystem = UserMetricsSystems
        .getMetricSystem(METRICS_NAMESPACE, builder => {
          builder.registerCounter(COUNTER_NAME)
        })

    driverMetricsSystem.counter(COUNTER_NAME).inc(5)
  }
}
```

The monitoring library includes a sample application that demonstrates how to use the

`UserMetricsSystem` class.

Send application logs using Log4j

To send your Azure Databricks application logs to Azure Log Analytics using the Log4j appender in the library, follow these steps:

1. Build the **spark-listeners-1.0-SNAPSHOT.jar** and the **spark-listeners-loganalytics-1.0-SNAPSHOT.jar** JAR file as described in the GitHub readme.

2. Create a **log4j.properties** configuration file for your application. Include the following configuration properties. Substitute your application package name and log level where indicated:

   ```
   log4j.appender.A1=com.microsoft.pnp.logging.loganalytics.LogAnalyticsAppender
   log4j.appender.A1.layout=com.microsoft.pnp.logging.JSONLayout
   log4j.appender.A1.layout.LocationInfo=false
   log4j.additivity.<your application package name>=false
   log4j.logger.<your application package name>=<log level>, A1
   ```

   You can find a sample configuration file here.

3. In your application code, include the **spark-listeners-loganalytics** project, and import `com.microsoft.pnp.logging.Log4jconfiguration` to your application code.

   ```
   import com.microsoft.pnp.logging.Log4jConfiguration
   ```

4. Configure Log4j using the **log4j.properties** file you created in step 3:

   ```
   getClass.getResourceAsStream("<path to file in your JAR file>/log4j.properties") {
         stream => {
           Log4jConfiguration.configure(stream)
         }
   }
   ```

5. Add Apache Spark log messages at the appropriate level in your code as required. For example, use the `logDebug` method to send a debug log message. For more information, see Logging in the Spark documentation.

   ```
   logTrace("Trace message")
   logDebug("Debug message")
   logInfo("Info message")
   logWarning("Warning message")
   logError("Error message")
   ```

Run the sample application

The monitoring library includes a sample application that demonstrates how to send both application metrics and application logs to Azure Monitor. To run the sample:

1. Build the **spark-jobs** project in the monitoring library, as described in the GitHub readme.

2. Navigate to your Databricks workspace and create a new job, as described here.

3. In the job detail page, select **Set JAR**.

4. Upload the JAR file from `/src/spark-jobs/target/spark-jobs-1.0-SNAPSHOT.jar`.

5. For **Main class**, enter `com.microsoft.pnp.samplejob.StreamingQueryListenerSampleJob`.
6. Select a cluster that is already configured to use the monitoring library. See Configure Azure Databricks to send metrics to Azure Monitor.

When the job runs, you can view the application logs and metrics in your Log Analytics workspace.

Application logs appear under SparkLoggingEvent_CL:

```
SparkLoggingEvent_CL | where logger_name_s contains "com.microsoft.pnp"
```

Application metrics appear under SparkMetric_CL:

```
SparkMetric_CL | where name_s contains "rowcounter" | limit 50
```

> **IMPORTANT**
> After you verify the metrics appear, stop the sample application job.

Next steps

Deploy the performance monitoring dashboard that accompanies this code library to troubleshoot performance issues in your production Azure Databricks workloads.

Use dashboards to visualize Azure Databricks metrics

Use dashboards to visualize Azure Databricks metrics

11/19/2021 • 7 minutes to read • Edit Online

This article shows how to set up a Grafana dashboard to monitor Azure Databricks jobs for performance issues.

Azure Databricks is a fast, powerful, and collaborative Apache Spark–based analytics service that makes it easy to rapidly develop and deploy big data analytics and artificial intelligence (AI) solutions. Monitoring is a critical component of operating Azure Databricks workloads in production. The first step is to gather metrics into a workspace for analysis. In Azure, the best solution for managing log data is Azure Monitor. Azure Databricks does not natively support sending log data to Azure monitor, but a library for this functionality is available in GitHub.

This library enables logging of Azure Databricks service metrics as well as Apache Spark structure streaming query event metrics. Once you've successfully deployed this library to an Azure Databricks cluster, you can further deploy a set of Grafana dashboards that you can deploy as part of your production environment.

Prerequisites

Configure your Azure Databricks cluster to use the monitoring library, as described in the GitHub readme.

Deploy the Azure Log Analytics workspace

To deploy the Azure Log Analytics workspace, follow these steps:

1. Navigate to the `/perftools/deployment/loganalytics` directory.
2. Deploy the **logAnalyticsDeploy.json** Azure Resource Manager template. For more information about deploying Resource Manager templates, see Deploy resources with Resource Manager templates and Azure CLI. The template has the following parameters:

- **location**: The region where the Log Analytics workspace and dashboards are deployed.
- **serviceTier**: The workspace pricing tier. See here for a list of valid values.
- **dataRetention** (optional): The number of days the log data is retained in the Log Analytics workspace. The default value is 30 days. If the pricing tier is `Free`, the data retention must be seven days.
- **workspaceName** (optional): A name for the workspace. If not specified, the template generates a name.

```
az group deployment create --resource-group <resource-group-name> --template-file
logAnalyticsDeploy.json --parameters location='East US' serviceTier='Standalone'
```

This template creates the workspace and also creates a set of predefined queries that are used by dashboard.

Deploy Grafana in a virtual machine

Grafana is an open source project you can deploy to visualize the time series metrics stored in your Azure Log Analytics workspace using the Grafana plugin for Azure Monitor. Grafana executes on a virtual machine (VM) and requires a storage account, virtual network, and other resources. To deploy a virtual machine with the bitnami-certified Grafana image and associated resources, follow these steps:

1. Use the Azure CLI to accept the Azure Marketplace image terms for Grafana.

   ```
   az vm image accept-terms --publisher bitnami --offer grafana --plan default
   ```

2. Navigate to the `/spark-monitoring/perftools/deployment/grafana` directory in your local copy of the GitHub repo.

3. Deploy the **grafanaDeploy.json** Resource Manager template as follows:

   ```
   export DATA_SOURCE="https://raw.githubusercontent.com/mspnp/spark-
   monitoring/master/perftools/deployment/grafana/AzureDataSource.sh"
   az group deployment create \
       --resource-group <resource-group-name> \
       --template-file grafanaDeploy.json \
       --parameters adminPass='<vm password>' dataSource=$DATA_SOURCE
   ```

Once the deployment is complete, the bitnami image of Grafana is installed on the virtual machine.

Update the Grafana password

As part of the setup process, the Grafana installation script outputs a temporary password for the **admin** user. You need this temporary password to sign in. To obtain the temporary password, follow these steps:

1. Log in to the Azure portal.
2. Select the resource group where the resources were deployed.
3. Select the VM where Grafana was installed. If you used the default parameter name in the deployment template, the VM name is prefaced with **sparkmonitoring-vm-grafana**.
4. In the **Support + troubleshooting** section, click **Boot diagnostics** to open the boot diagnostics page.
5. Click **Serial log** on the boot diagnostics page.
6. Search for the following string: "Setting Bitnami application password to".
7. Copy the password to a safe location.

Next, change the Grafana administrator password by following these steps:

1. In the Azure portal, select the VM and click **Overview**.
2. Copy the public IP address.
3. Open a web browser and navigate to the following URL: `http://<IP address>:3000`.
4. At the Grafana login screen, enter **admin** for the user name, and use the Grafana password from the previous steps.
5. Once logged in, select **Configuration** (the gear icon).
6. Select **Server Admin**.
7. On the **Users** tab, select the **admin** login.
8. Update the password.

Create an Azure Monitor data source

1. Create a service principal that allows Grafana to manage access to your Log Analytics workspace. For more information, see Create an Azure service principal with Azure CLI

   ```
   az ad sp create-for-rbac --name http://<service principal name> --role "Log Analytics Reader"
   ```

2. Note the values for appId, password, and tenant in the output from this command:

   ```
   {
       "appId": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
       "displayName": "azure-cli-2019-03-27-00-33-39",
       "name": "http://<service principal name>",
       "password": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
       "tenant": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
   }
   ```

3. Log into Grafana as described earlier. Select **Configuration** (the gear icon) and then **Data Sources**.
4. In the **Data Sources** tab, click **Add data source**.
5. Select **Azure Monitor** as the data source type.
6. In the **Settings** section, enter a name for the data source in the **Name** textbox.
7. In the **Azure Monitor API Details** section, enter the following information:
   - Subscription Id: Your Azure subscription ID.
   - Tenant Id: The tenant ID from earlier.
   - Client Id: The value of "appId" from earlier.
   - Client Secret: The value of "password" from earlier.
8. In the **Azure Log Analytics API Details** section, check the **Same Details as Azure Monitor API** checkbox.
9. Click **Save & Test**. If the Log Analytics data source is correctly configured, a success message is displayed.

Create the dashboard

Create the dashboards in Grafana by following these steps:

1. Navigate to the `/perftools/dashboards/grafana` directory in your local copy of the GitHub repo.
2. Run the following script:

```
export WORKSPACE=<your Azure Log Analytics workspace ID>
export LOGTYPE=SparkListenerEvent_CL

sh DashGen.sh
```

The output from the script is a file named **SparkMonitoringDash.json**.

3. Return to the Grafana dashboard and select **Create** (the plus icon).
4. Select **Import**.
5. Click **Upload .json File**.
6. Select the **SparkMonitoringDash.json** file created in step 2.
7. In the **Options** section, under **ALA**, select the Azure Monitor data source created earlier.
8. Click **Import**.

Visualizations in the dashboards

Both the Azure Log Analytics and Grafana dashboards include a set of time-series visualizations. Each graph is time-series plot of metric data related to an Apache Spark job, stages of the job, and tasks that make up each stage.

The visualizations are:

Job latency

This visualization shows execution latency for a job, which is a coarse view on the overall performance of a job. Displays the job execution duration from start to completion. Note that the job start time is not the same as the job submission time. Latency is represented as percentiles (10%, 30%, 50%, 90%) of job execution indexed by cluster ID and application ID.

Stage latency

The visualization shows the latency of each stage per cluster, per application, and per individual stage. This visualization is useful for identifying a particular stage that is running slowly.

Task latency

This visualization shows task execution latency. Latency is represented as a percentile of task execution per cluster, stage name, and application.

Sum Task Execution per host

This visualization shows the sum of task execution latency per host running on a cluster. Viewing task execution latency per host identifies hosts that have much higher overall task latency than other hosts. This may mean that tasks have been inefficiently or unevenly distributed to hosts.

Task metrics

This visualization shows a set of the execution metrics for a given task's execution. These metrics include the size and duration of a data shuffle, duration of serialization and deserialization operations, and others. For the full set of metrics, view the Log Analytics query for the panel. This visualization is useful for understanding the operations that make up a task and identifying resource consumption of each operation. Spikes in the graph represent costly operations that should be investigated.

Cluster throughput

This visualization is a high-level view of work items indexed by cluster and application to represent the amount of work done per cluster and application. It shows the number of jobs, tasks, and stages completed per cluster, application, and stage in one minute increments.

Streaming Throughput/Latency

This visualization is related to the metrics associated with a structured streaming query. The graph shows the number of input rows per second and the number of rows processed per second. The streaming metrics are also represented per application. These metrics are sent when the OnQueryProgress event is generated as the structured streaming query is processed and the visualization represents streaming latency as the amount of time, in milliseconds, taken to execute a query batch.

Resource consumption per executor

Next is a set of visualizations for the dashboard show the particular type of resource and how it is consumed per executor on each cluster. These visualizations help identify outliers in resource consumption per executor. For example, if the work allocation for a particular executor is skewed, resource consumption will be elevated in relation to other executors running on the cluster. This can be identified by spikes in the resource consumption for an executor.

Executor compute time metrics

Next is a set of visualizations for the dashboard that show the ratio of executor serialize time, deserialize time, CPU time, and Java virtual machine time to overall executor compute time. This demonstrates visually how much each of these four metrics is contributing to overall executor processing.

Shuffle metrics

The final set of visualizations shows the data shuffle metrics associated with a structured streaming query across all executors. These include shuffle bytes read, shuffle bytes written, shuffle memory, and disk usage in queries where the file system is used.

Next steps

Troubleshoot performance bottlenecks

Troubleshoot performance bottlenecks in Azure Databricks

11/19/2021 • 6 minutes to read • Edit Online

This article describes how to use monitoring dashboards to find performance bottlenecks in Spark jobs on Azure Databricks.

Azure Databricks is an Apache Spark–based analytics service that makes it easy to rapidly develop and deploy big data analytics. Monitoring and troubleshooting performance issues is a critical when operating production Azure Databricks workloads. To identify common performance issues, it's helpful to use monitoring visualizations based on telemetry data.

Prerequisites

To set up the Grafana dashboards shown in this article:

- Configure your Databricks cluster to send telemetry to a Log Analytics workspace, using the Azure Databricks Monitoring Library. For details, see the GitHub readme.
- Deploy Grafana in a virtual machine. See Use dashboards to visualize Azure Databricks metrics.

The Grafana dashboard that is deployed includes a set of time-series visualizations. Each graph is time-series plot of metrics related to an Apache Spark job, the stages of the job, and tasks that make up each stage.

Azure Databricks performance overview

Azure Databricks is based on Apache Spark, a general-purpose distributed computing system. Application code, known as a **job**, executes on an Apache Spark cluster, coordinated by the cluster manager. In general, a job is the highest-level unit of computation. A job represents the complete operation performed by the Spark application. A typical operation includes reading data from a source, applying data transformations, and writing the results to storage or another destination.

Jobs are broken down into **stages**. The job advances through the stages sequentially, which means that later stages must wait for earlier stages to complete. Stages contain groups of identical **tasks** that can be executed in parallel on multiple nodes of the Spark cluster. Tasks are the most granular unit of execution taking place on a subset of the data.

The next sections describe some dashboard visualizations that are useful for performance troubleshooting.

Job and stage latency

Job latency is the duration of a job execution from when it starts until it completes. It is shown as percentiles of a job execution per cluster and application ID, to allow the visualization of outliers. The following graph shows a job history where the 90th percentile reached 50 seconds, even though the 50th percentile was consistently around 10 seconds.

Investigate job execution by cluster and application, looking for spikes in latency. Once clusters and applications with high latency are identified, move on to investigate stage latency.

Stage latency is also shown as percentiles to allow the visualization of outliers. Stage latency is broken out by cluster, application, and stage name. Identify spikes in task latency in the graph to determine which tasks are holding back completion of the stage.

The cluster throughput graph shows the number of jobs, stages, and tasks completed per minute. This helps you to understand the workload in terms of the relative number of stages and tasks per job. Here you can see that the number of jobs per minute ranges between 2 and 6, while the number of stages is about 12 – 24 per minute.

Sum of task execution latency

This visualization shows the sum of task execution latency per host running on a cluster. Use this graph to detect tasks that run slowly due to the host slowing down on a cluster, or a misallocation of tasks per executor. In the following graph, most of the hosts have a sum of about 30 seconds. However, two of the hosts have sums that hover around 10 minutes. Either the hosts are running slow or the number of tasks per executor is misallocated.

The number of tasks per executor shows that two executors are assigned a disproportionate number of tasks, causing a bottleneck.

Task metrics per stage

The task metrics visualization gives the cost breakdown for a task execution. You can use it see the relative time spent on tasks such as serialization and deserialization. This data might show opportunities to optimize — for example, by using broadcast variables to avoid shipping data. The task metrics also show the shuffle data size for a task, and the shuffle read and write times. If these values are high, it means that a lot of data is moving across the network.

Another task metric is the scheduler delay, which measures how long it takes to schedule a task. Ideally, this value should be low compared to the executor compute time, which is the time spent actually executing the task.

The following graph shows a scheduler delay time (3.7 s) that exceeds the executor compute time (1.1 s). That means more time is spent waiting for tasks to be scheduled than doing the actual work.

In this case, the problem was caused by having too many partitions, which caused a lot of overhead. Reducing the number of partitions lowered the scheduler delay time. The next graph shows that most of the time is spent executing the task.

Streaming throughput and latency

Streaming throughput is directly related to structured streaming. There are two important metrics associated with streaming throughput: Input rows per second and processed rows per second. If input rows per second outpaces processed rows per second, it means the stream processing system is falling behind. Also, if the input data comes from Event Hubs or Kafka, then input rows per second should keep up with the data ingestion rate at the front end.

Two jobs can have similar cluster throughput but very different streaming metrics. The following screenshot shows two different workloads. They are similar in terms of cluster throughput (jobs, stages, and tasks per minute). But the second run processes 12,000 rows/sec versus 4,000 rows/sec.

Streaming throughput is often a better business metric than cluster throughput, because it measures the number of data records that are processed.

Resource consumption per executor

These metrics help to understand the work that each executor performs.

Percentage metrics measure how much time an executor spends on various things, expressed as a ratio of time spent versus the overall executor compute time. The metrics are:

- % Serialize time
- % Deserialize time
- % CPU executor time
- % JVM time

These visualizations show how much each of these metrics contributes to overall executor processing.

Shuffle metrics are metrics related to data shuffling across the executors.

- Shuffle I/O
- Shuffle memory
- File system usage

- Disk usage

Common performance bottlenecks

Two common performance bottlenecks in Spark are *task stragglers* and a *non-optimal shuffle partition count*.

Task stragglers

The stages in a job are executed sequentially, with earlier stages blocking later stages. If one task executes a shuffle partition more slowly than other tasks, all tasks in the cluster must wait for the slow task to catch up before the stage can end. This can happen for the following reasons:

1. A host or group of hosts are running slow. Symptoms: High task, stage, or job latency and low cluster throughput. The summation of tasks latencies per host won't be evenly distributed. However, resource consumption will be evenly distributed across executors.

2. Tasks have an expensive aggregation to execute (data skewing). Symptoms: High task latency, high stage latency, high job latency, or low cluster throughput, but the summation of latencies per host is evenly distributed. Resource consumption will be evenly distributed across executors.

3. If partitions are of unequal size, a larger partition may cause unbalanced task execution (partition skewing). Symptoms: Executor resource consumption is high compared to other executors running on the cluster. All tasks running on that executor will run slow and hold the stage execution in the pipeline. Those stages are said to be *stage barriers*.

Non-optimal shuffle partition count

During a structured streaming query, the assignment of a task to an executor is a resource-intensive operation for the cluster. If the shuffle data isn't the optimal size, the amount of delay for a task will negatively impact throughput and latency. If there are too few partitions, the cores in the cluster will be underutilized which can result in processing inefficiency. Conversely, if there are too many partitions, there's a great deal of management overhead for a small number of tasks.

Use the resource consumption metrics to troubleshoot partition skewing and misallocation of executors on the cluster. If a partition is skewed, executor resources will be elevated in comparison to other executors running on the cluster.

For example, the following graph shows that the memory used by shuffling on the first two executors is 90X bigger than the other executors:

Run Apache Cassandra on Azure VMs

11/19/2021 • 9 minutes to read • Edit Online

This article describes performance considerations for running Apache Cassandra on Azure virtual machines.

These recommendations are based on the results of performance tests, which you can find on GitHub. You should use these recommendations as a baseline and then test against your own workload.

Azure Managed Instance for Apache Cassandra

If you're looking for a more automated service for running Apache Cassandra on Azure virtual machines, consider using Azure Managed Instance for Apache Cassandra. This service automates the deployment, management (patching and node health), and scaling of nodes within an Apache Cassandra cluster. It also provides the capability for hybrid clusters, so Apache Cassandra datacenters deployed in Azure can join an existing on-premises or third-party hosted Cassandra ring. The service is deployed by using Azure virtual machine scale sets. The following recommendations were adopted during the development of this service.

Azure VM sizes and disk types

Cassandra workloads on Azure commonly use either Standard_DS14_v2 or Standard_DS13_v2 virtual machines. Cassandra workloads benefit from having more memory in the VM, so consider memory optimized virtual machine sizes, such as Standard_DS14_v2, or local-storage optimized sizes such as Standard_L16s_v2.

For durability, data and commit logs are commonly stored on a stripe set of two to four 1-TB premium managed disks (P30).

Cassandra nodes should not be too data-dense. We recommend having at most 1 – 2 TB of data per VM and enough free space for compaction. To achieve the highest possible combined throughput and IOPS using premium managed disks, we recommend creating a stripe set from a few 1-TB disks, instead of using a single 2-TB or 4-TB disk. For example, on a DS14_v2 VM, four 1-TB disks have a maximum IOPS of 4 × 5000 = 20 K, versus 7.5 K for a single 4-TB disk.

As Azure Ultra Disks become more widely available across regions, evaluate them for Cassandra workloads that need smaller disk capacity. They can provide higher IOPS/throughput and lower latency on VM sizes like Standard_D32s_v3 and Standard_D16s_v3.

For Cassandra workloads that don't need durable storage — that is, where data can be easily reconstructed from another storage medium — consider using Standard_L32s_v2 or Standard_L16s_v2 VMs. These VMs sizes have large and fast local *temporary* NVMe disks.

For more information, see Comparing performance of Azure local/ephemeral vs attached/persistent disks (GitHub).

Accelerated Networking

Cassandra nodes make heavy use of the network to send and receive data from the client VM and to communicate between nodes for replication. For optimal performance, Cassandra VMs benefit from high-throughput and low-latency network.

We recommended enabling Accelerated Networking on the NIC of the Cassandra node and on VMs running client applications accessing Cassandra.

Accelerated networking requires a modern Linux distribution with the latest drivers, such as Cent OS 7.5+ or

Ubuntu 16.x/18.x. For more information, see Create a Linux virtual machine with Accelerated Networking.

Azure VM data disk caching

Cassandra read workloads perform best when random-access disk latency is low. We recommend using Azure managed disks with ReadOnly caching enabled. ReadOnly caching provides lower average latency, because the data is read from the cache on the host instead of going to the backend storage.

Read-heavy, random-read workloads like Cassandra benefit from the lower read latency even though cached mode has lower throughput limits than uncached mode. (For example, DS14_v2 virtual machines have a maximum cached throughput of 512 MB/s versus uncached of 768 MB/s.)

ReadOnly caching is particularly helpful for Cassandra time-series and other workloads where the working dataset fits in the host cache and data is not constantly overwritten. For example, DS14_v2 provides a cache size of 512 GB, which could store up to 50% of the data from a Cassandra node with 1-2 TB data density.

There is no significant performance penalty from cache-misses when ReadOnly caching is enabled, so cached mode is recommended for all but the most write-heavy workloads.

For more information, see Comparing Azure VM data disk caching configurations (GitHub).

Linux read-ahead

In most Linux distributions in the Azure Marketplace, the default block device read-ahead setting is 4096 KB. Cassandra's read IOs are usually random and relatively small. Therefore, having a large read-ahead wastes throughput by reading parts of files that aren't needed.

To minimize unnecessary lookahead, set the Linux block device read-ahead setting to 8 KB. (See Recommended production settings in the DataStax documentation.)

Configure 8 KB read-ahead for all block devices in the stripe set and on the array device itself (for example, `/dev/md0`).

For more information, see Comparing impact of disk read-ahead settings (GitHub).

Disk array mdadm chunk size

When running Cassandra on Azure, it's common to create an mdadm stripe set (that is, RAID 0) of multiple data disks to increase the overall disk throughput and IOPS closer to the VM limits. Optimal disk stripe size is an application-specific setting. For example, for SQL Server OLTP workloads, the recommendation is 64 KB. For data warehousing workloads, the recommendation is 256 KB.

Our tests found no significant difference between chunk sizes of 64k, 128k, and 256k for Cassandra read workloads. There seems to be a small, slightly noticeable, advantage to the 128k chunk size. Therefore, we recommend the following:

- If you're already using a chunk size of 64 K or 256 K, it doesn't make sense rebuild the disk array to use 128 K size.
- For a new configuration, it makes senses to use 128 K from the beginning.

For more information, see Measuring impact of mdadm chunk sizes on Cassandra performance (GitHub).

Commit log filesystem

Cassandra writes perform best when commit logs are on disks with high throughput and low latency. In the default configuration, Cassandra 3.x flushes data from memory to the commit log file every ~10 seconds and

doesn't touch the disk for every write. In this configuration, write performance is almost identical whether the commit log is on premium attached disks versus local/ephemeral disks.

Commit logs must be durable, so that a restarted node can reconstruct any data not yet in data files from the flushed commit logs. For better durability, store commit logs on premium managed disks and not on local storage, which can be lost if the VM is migrated to another host.

Based on our tests, Cassandra on CentOS 7.x may have *lower* write performance when commit logs are on the xfs versus ext4 filesystem. Turning on commit log compression brings xfs performance in line with ext4. Compressed xfs performs as well as compressed and non-compressed ext4 in our tests.

For more information, see Observations on ext4 and xfs file systems and compressed commit logs (GitHub).

Measuring baseline VM performance

After deploying the VMs for the Cassandra ring, run a few synthetic tests to establish baseline network and disk performance. Use these tests to confirm that performance is in line with expectations, based on the VM size.

Later, when you run the actual workload, knowing the performance baseline makes it easier to investigate potential bottlenecks. For example, knowing the baseline performance for network egress on the VM can help to rule out network as a bottleneck.

For more information about running performance tests, see Validating baseline Azure VM Performance (GitHub).

Document size

Cassandra read and write performance depends on the document size. You can expect to see higher latency and lower operations/second when reading or writing with larger documents.

For more information, see Comparing relative performance of various Cassandra document sizes (GitHub).

Replication factor

Most Cassandra workloads use a replication factor (RF) of 3 when using attached premium disks and even 5 when using temporary/ephemeral local disks. The number of nodes in the Cassandra ring should be a multiple of the replication factor. For example, RF 3 implies a ring of 3, 6, 9, or 12 nodes, while RF 5 would have 5, 10, 15, or 20 nodes. When using RF greater than 1 and a consistency level of LOCAL_QUORUM, it's normal for read and write performance to be lower than the same workload running with RF 1.

For more information, see Comparing relative performance of various replication factors (GitHub).

Linux page caching

When reading data files, Cassandra's Java code uses regular file I/O and benefits from Linux page caching. After parts of the file are read one time, the read content is stored in the OS page cache. Subsequent read access to the same data is much faster.

For this reason, when executing read performance tests against the same data, the second and subsequent reads will appear to be much faster than the original read, which needed to access data on the remote data disk or from the host cache when ReadOnly is enabled. To get similar performance measurements on subsequent runs, clear the Linux page cache and restart the Cassandra service to clear its internal memory. When ReadOnly caching is enabled, the data might be in the host cache, and subsequent reads will be faster even after clearing the OS page cache and restarting the Cassandra service.

For more information, see Observations on Cassandra usage of Linux page caching (GitHub).

Multi-datacenter replication

Cassandra natively supports the concept of multiple data centers, making it easy to configure one Cassandra

ring across multiple Azure regions or across availability zones within one region.

For a multiregion deployment, use Azure Global VNet-peering to connect the virtual networks in the different regions. When VMs are deployed in the same region but in separate availability zones, the VMs can be in the same virtual network.

It's important to measure the baseline roundtrip latency between regions. Network latency between regions can be 10-100 times higher than latency within a region. Expect a lag between data appearing in the second region when using LOCAL_QUORUM write consistency, or significantly decreased performance of writes when using EACH_QUORUM.

When running Apache Cassandra at scale, and specifically in a multi-DC environment, node repair becomes challenging. Tools such as Reaper can help to coordinate repairs at scale (for example, across all the nodes in a data center, one data center at a time, to limit the load on the whole cluster). However, node repair for large clusters is not yet a fully solved problem and applies in all environments, whether on-premises or in the cloud.

When nodes are added to a secondary region, performance will not scale linearly, because some bandwidth and CPU/disk resources are spent on receiving and sending replication traffic across regions.

For more information, see Measuring impact of multi-dc cross-region replication (GitHub).

Hinted-handoff configuration

In a multiregion Cassandra ring, write workloads with consistency level of LOCAL_QUORUM may lose data in the secondary region. By default, Cassandra hinted handoff is throttled to a relatively low maximum throughput and three-hour hint lifetime. For workloads with heavy writes, we recommended increasing the hinted handoff throttle and hint window time to ensure hints are not dropped before they are replicated.

For more information, see Observations on hinted handoff in cross-region replication (GitHub).

Next steps

For more information about these performance results, see Cassandra on Azure VMs Performance Experiments.

For information on general Cassandra settings, not specific to Azure, see:

- DataStax Recommended Production Settings
- Apache Cassandra Hardware Choices
- Apache Cassandra Configuration File

The following reference architecture deploys Cassandra as part of an n-tier configuration:

- Linux N-tier application in Azure with Apache Cassandra

Build microservices on Azure

11/19/2021 • 5 minutes to read • Edit Online

Microservices are a popular architectural style for building applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. But a successful microservices architecture requires a different approach to designing and building applications.

A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.

What are microservices?

- Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
- Each service is a separate codebase, which can be managed by a small development team.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.

Besides for the services themselves, some other components appear in a typical microservices architecture:

Management/orchestration. This component is responsible for placing services on nodes, identifying failures, rebalancing services across nodes, and so forth. Typically this component is an off-the-shelf technology such as Kubernetes, rather than something custom built.

API Gateway. The API gateway is the entry point for clients. Instead of calling services directly, clients call the API gateway, which forwards the call to the appropriate services on the back end.

Advantages of using an API gateway include:

- It decouples clients from services. Services can be versioned or refactored without needing to update all of the clients.
- Services can use messaging protocols that are not web friendly, such as AMQP.
- The API Gateway can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.
- Out-of-the-box policies, like for throttling, caching, transformation, or validation.

Benefits

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process. New features may be held up waiting for a bug fix to be integrated, tested, and published.
- **Small, focused teams**. A microservice should be small enough that a single feature team can build, test, and deploy it. Small team sizes promote greater agility. Large teams tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.
- **Small code base**. In a monolithic application, there is a tendency over time for code dependencies to become tangled. Adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features.
- **Mix of technologies**. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Fault isolation**. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).
- **Scalability**. Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire application. Using an orchestrator such as Kubernetes or Service Fabric, you can pack a higher density of services onto a single host, which allows for more efficient utilization of resources.
- **Data isolation**. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

- **Complexity**. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing**. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.
- **Lack of governance**. The decentralized approach to building microservices has advantages, but it can

also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

- **Network congestion and latency**. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns like queue-based load leveling.
- **Data integrity**. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.
- **Management**. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.
- **Versioning**. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.
- **Skill set**. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

Process for building a microservices architecture

The articles listed here present a structured approach for designing, building, and operating a microservices architecture.

Domain analysis. To avoid some common pitfalls when designing microservices, use domain analysis to define your microservice boundaries. Follow these steps:

1. Use domain analysis to model microservices.
2. Use tactical DDD to design microservices.
3. Identify microservice boundaries.

Design the services. Microservices require a different approach to designing and building applications. For more information, see Designing a microservices architecture.

Operate in production. Because microservices architectures are distributed, you must have robust operations for deployment and monitoring.

- CI/CD for microservices architectures
- Build a CI/CD pipeline for microservices on Kubernetes
- Monitor microservices running on Azure Kubernetes Service (AKS)

Microservices reference architectures for Azure

- Microservices architecture on Azure Kubernetes Service (AKS)
- Microservices architecture on Azure Service Fabric

Using domain analysis to model microservices

11/19/2021 • 8 minutes to read • Edit Online

One of the biggest challenges of microservices is to define the boundaries of individual services. The general rule is that a service should do "one thing" — but putting that rule into practice requires careful thought. There is no mechanical process that will produce the "right" design. You have to think deeply about your business domain, requirements, and goals. Otherwise, you can end up with a haphazard design that exhibits some undesirable characteristics, such as hidden dependencies between services, tight coupling, or poorly designed interfaces. This article shows a domain-driven approach to designing microservices.

This article uses a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation here.

Introduction

Microservices should be designed around business capabilities, not horizontal layers such as data access or messaging. In addition, they should have loose coupling and high functional cohesion. Microservices are *loosely coupled* if you can change one service without requiring other services to be updated at the same time. A microservice is *cohesive* if it has a single, well-defined purpose, such as managing user accounts or tracking delivery history. A service should encapsulate domain knowledge and abstract that knowledge from clients. For example, a client should be able to schedule a drone without knowing the details of the scheduling algorithm or how the drone fleet is managed.

Domain-driven design (DDD) provides a framework that can get you most of the way to a set of well-designed microservices. DDD has two distinct phases, strategic and tactical. In strategic DDD, you are defining the large-scale structure of the system. Strategic DDD helps to ensure that your architecture remains focused on business capabilities. Tactical DDD provides a set of design patterns that you can use to create the domain model. These patterns include entities, aggregates, and domain services. These tactical patterns will help you to design microservices that are both loosely coupled and cohesive.

In this article and the next, we'll walk through the following steps, applying them to the Drone Delivery application:

1. Start by analyzing the business domain to understand the application's functional requirements. The output of this step is an informal description of the domain, which can be refined into a more formal set of domain models.
2. Next, define the *bounded contexts* of the domain. Each bounded context contains a domain model that represents a particular subdomain of the larger application.
3. Within a bounded context, apply tactical DDD patterns to define entities, aggregates, and domain services.
4. Use the results from the previous step to identify the microservices in your application.

In this article, we cover the first three steps, which are primarily concerned with DDD. In the next article, we'll identify the microservices. However, it's important to remember that DDD is an iterative, ongoing process.

Service boundaries aren't fixed in stone. As an application evolves, you may decide to break apart a service into several smaller services.

> **NOTE**
> This article doesn't show a complete and comprehensive domain analysis. We deliberately kept the example brief, to illustrate the main points. For more background on DDD, we recommend Eric Evans' *Domain-Driven Design*, the book that first introduced the term. Another good reference is *Implementing Domain-Driven Design* by Vaughn Vernon.

Scenario: Drone delivery

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

Analyze the domain

Using a DDD approach will help you to design microservices so that every service forms a natural fit to a functional business requirement. It can help you to avoid the trap of letting organizational boundaries or technology choices dictate your design.

Before writing any code, you need a bird's eye view of the system that you are creating. DDD starts by modeling the business domain and creating a *domain model*. The domain model is an abstract model of the business domain. It distills and organizes domain knowledge, and provides a common language for developers and domain experts.

Start by mapping all of the business functions and their connections. This will likely be a collaborative effort that involves domain experts, software architects, and other stakeholders. You don't need to use any particular formalism. Sketch a diagram or draw on whiteboard.

As you fill in the diagram, you may start to identify discrete subdomains. Which functions are closely related? Which functions are core to the business, and which provide ancillary services? What is the dependency graph? During this initial phase, you aren't concerned with technologies or implementation details. That said, you should note the place where the application will need to integrate with external systems, such as CRM, payment processing, or billing systems.

Example: Drone delivery application

After some initial domain analysis, the Fabrikam team came up with a rough sketch that depicts the Drone Delivery domain.

- **Shipping** is placed in the center of the diagram, because it's core to the business. Everything else in the diagram exists to enable this functionality.
- **Drone management** is also core to the business. Functionality that is closely related to drone management

includes **drone repair** and using **predictive analysis** to predict when drones need servicing and maintenance.

- **ETA analysis** provides time estimates for pickup and delivery.
- **Third-party transportation** will enable the application to schedule alternative transportation methods if a package cannot be shipped entirely by drone.
- **Drone sharing** is a possible extension of the core business. The company may have excess drone capacity during certain hours, and could rent out drones that would otherwise be idle. This feature will not be in the initial release.
- **Video surveillance** is another area that the company might expand into later.
- **User accounts**, **Invoicing**, and **Call center** are subdomains that support the core business.

Notice that at this point in the process, we haven't made any decisions about implementation or technologies. Some of the subsystems may involve external software systems or third-party services. Even so, the application needs to interact with these systems and services, so it's important to include them in the domain model.

> **NOTE**
>
> When an application depends on an external system, there is a risk that the external system's data schema or API will leak into your application, ultimately compromising the architectural design. This is particularly true with legacy systems that may not follow modern best practices, and may use convoluted data schemas or obsolete APIs. In that case, it's important to have a well-defined boundary between these external systems and the application. Consider using the Strangler Fig pattern or the Anti-Corruption Layer pattern for this purpose.

Define bounded contexts

The domain model will include representations of real things in the world — users, drones, packages, and so forth. But that doesn't mean that every part of the system needs to use the same representations for the same things.

For example, subsystems that handle drone repair and predictive analysis will need to represent many physical characteristics of drones, such as their maintenance history, mileage, age, model number, performance characteristics, and so on. But when it's time to schedule a delivery, we don't care about those things. The scheduling subsystem only needs to know whether a drone is available, and the ETA for pickup and delivery.

If we tried to create a single model for both of these subsystems, it would be unnecessarily complex. It would also become harder for the model to evolve over time, because any changes will need to satisfy multiple teams working on separate subsystems. Therefore, it's often better to design separate models that represent the same real-world entity (in this case, a drone) in two different contexts. Each model contains only the features and attributes that are relevant within its particular context.

This is where the DDD concept of *bounded contexts* comes into play. A bounded context is simply the boundary within a domain where a particular domain model applies. Looking at the previous diagram, we can group functionality according to whether various functions will share a single domain model.

Bounded contexts are not necessarily isolated from one another. In this diagram, the solid lines connecting the bounded contexts represent places where two bounded contexts interact. For example, Shipping depends on User Accounts to get information about customers, and on Drone Management to schedule drones from the fleet.

In the book *Domain Driven Design*, Eric Evans describes several patterns for maintaining the integrity of a domain model when it interacts with another bounded context. One of the main principles of microservices is that services communicate through well-defined APIs. This approach corresponds to two patterns that Evans

calls Open Host Service and Published Language. The idea of Open Host Service is that a subsystem defines a formal protocol (API) for other subsystems to communicate with it. Published Language extends this idea by publishing the API in a form that other teams can use to write clients. In the article Designing APIs for microservices, we discuss using OpenAPI Specification (formerly known as Swagger) to define language-agnostic interface descriptions for REST APIs, expressed in JSON or YAML format.

For the rest of this journey, we will focus on the Shipping bounded context.

Next steps

After completing a domain analysis, the next step is to apply tactical DDD, to define your domain models with more precision.

Tactical DDD

Using tactical DDD to design microservices

11/19/2021 • 5 minutes to read • Edit Online

During the strategic phase of DDD, you are mapping out the business domain and defining bounded contexts for your domain models. Tactical DDD is when you define your domain models with more precision. The tactical patterns are applied within a single bounded context. In a microservices architecture, we are particularly interested in the entity and aggregate patterns. Applying these patterns will help us to identify natural boundaries for the services in our application (see the next article in this series). As a general principle, a microservice should be no smaller than an aggregate, and no larger than a bounded context. First, we'll review the tactical patterns. Then we'll apply them to the Shipping bounded context in the Drone Delivery application.

Overview of the tactical patterns

This section provides a brief summary of the tactical DDD patterns, so if you are already familiar with DDD, you can probably skip this section. The patterns are described in more detail in chapters 5 – 6 of Eric Evans' book, and in *Implementing Domain-Driven Design* by Vaughn Vernon.

Entities. An entity is an object with a unique identity that persists over time. For example, in a banking application, customers and accounts would be entities.

- An entity has a unique identifier in the system, which can be used to look up or retrieve the entity. That doesn't mean the identifier is always exposed directly to users. It could be a GUID or a primary key in a database.
- An identity may span multiple bounded contexts, and may endure beyond the lifetime of the application. For example, bank account numbers or government-issued IDs are not tied to the lifetime of a particular application.
- The attributes of an entity may change over time. For example, a person's name or address might change, but they are still the same person.
- An entity can hold references to other entities.

Value objects. A value object has no identity. It is defined only by the values of its attributes. Value objects are also immutable. To update a value object, you always create a new instance to replace the old one. Value objects can have methods that encapsulate domain logic, but those methods should have no side-effects on the object's state. Typical examples of value objects include colors, dates and times, and currency values.

Aggregates. An aggregate defines a consistency boundary around one or more entities. Exactly one entity in an aggregate is the root. Lookup is done using the root entity's identifier. Any other entities in the aggregate are children of the root, and are referenced by following pointers from the root.

The purpose of an aggregate is to model transactional invariants. Things in the real world have complex webs of

relationships. Customers create orders, orders contain products, products have suppliers, and so on. If the application modifies several related objects, how does it guarantee consistency? How do we keep track of invariants and enforce them?

Traditional applications have often used database transactions to enforce consistency. In a distributed application, however, that's often not feasible. A single business transaction may span multiple data stores, or may be long running, or may involve third-party services. Ultimately it's up to the application, not the data layer, to enforce the invariants required for the domain. That's what aggregates are meant to model.

> **NOTE**
> An aggregate might consist of a single entity, without child entities. What makes it an aggregate is the transactional boundary.

Domain and application services. In DDD terminology, a service is an object that implements some logic without holding any state. Evans distinguishes between *domain services*, which encapsulate domain logic, and *application services*, which provide technical functionality, such as user authentication or sending an SMS message. Domain services are often used to model behavior that spans multiple entities.

> **NOTE**
> The term *service* is overloaded in software development. The definition here is not directly related to microservices.

Domain events. Domain events can be used to notify other parts of the system when something happens. As the name suggests, domain events should mean something within the domain. For example, "a record was inserted into a table" is not a domain event. "A delivery was cancelled" is a domain event. Domain events are especially relevant in a microservices architecture. Because microservices are distributed and don't share data stores, domain events provide a way for microservices to coordinate with each other. The article Interservice communication discusses asynchronous messaging in more detail.

There are a few other DDD patterns not listed here, including factories, repositories, and modules. These can be useful patterns for when you are implementing a microservice, but they are less relevant when designing the boundaries between microservice.

Drone delivery: Applying the patterns

We start with the scenarios that the Shipping bounded context must handle.

- A customer can request a drone to pick up goods from a business that is registered with the drone delivery service.
- The sender generates a tag (barcode or RFID) to put on the package.
- A drone will pick up and deliver a package from the source location to the destination location.
- When a customer schedules a delivery, the system provides an ETA based on route information, weather conditions, and historical data.
- When the drone is in flight, a user can track the current location and the latest ETA.
- Until a drone has picked up the package, the customer can cancel a delivery.
- The customer is notified when the delivery is completed.
- The sender can request delivery confirmation from the customer, in the form of a signature or finger print.
- Users can look up the history of a completed delivery.

From these scenarios, the development team identified the following **entities**.

- Delivery

- Package
- Drone
- Account
- Confirmation
- Notification
- Tag

The first four, Delivery, Package, Drone, and Account, are all **aggregates** that represent transactional consistency boundaries. Confirmations and Notifications are child entities of Deliveries, and Tags are child entities of Packages.

The **value objects** in this design include Location, ETA, PackageWeight, and PackageSize.

To illustrate, here is a UML diagram of the Delivery aggregate. Notice that it holds references to other aggregates, including Account, Package, and Drone.

There are two domain events:

- While a drone is in flight, the Drone entity sends DroneStatus events that describe the drone's location and status (in-flight, landed).
- The Delivery entity sends DeliveryTracking events whenever the stage of a delivery changes. These include DeliveryCreated, DeliveryRescheduled, DeliveryHeadedToDropoff, and DeliveryCompleted.

Notice that these events describe things that are meaningful within the domain model. They describe something about the domain, and aren't tied to a particular programming language construct.

The development team identified one more area of functionality, which doesn't fit neatly into any of the entities described so far. Some part of the system must coordinate all of the steps involved in scheduling or updating a delivery. Therefore, the development team added two **domain services** to the design: a *Scheduler* that coordinates the steps, and a *Supervisor* that monitors the status of each step, in order to detect whether any steps have failed or timed out. This is a variation of the Scheduler Agent Supervisor pattern.

Next steps

The next step is to define the boundaries for each microservice.

Identify microservice boundaries

Identifying microservice boundaries

11/19/2021 • 5 minutes to read • Edit Online

What is the right size for a microservice? You often hear something to the effect of, "not too big and not too small" — and while that's certainly correct, it's not very helpful in practice. But if you start from a carefully designed domain model, it's much easier to reason about microservices.

This article uses a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation here.

From domain model to microservices

In the previous article, we defined a set of bounded contexts for a Drone Delivery application. Then we looked more closely at one of these bounded contexts, the Shipping bounded context, and identified a set of entities, aggregates, and domain services for that bounded context.

Now we're ready to go from domain model to application design. Here's an approach that you can use to derive microservices from the domain model.

1. Start with a bounded context. In general, the functionality in a microservice should not span more than one bounded context. By definition, a bounded context marks the boundary of a particular domain model. If you find that a microservice mixes different domain models together, that's a sign that you may need to go back and refine your domain analysis.

2. Next, look at the aggregates in your domain model. Aggregates are often good candidates for microservices. A well-designed aggregate exhibits many of the characteristics of a well-designed microservice, such as:

   - An aggregate is derived from business requirements, rather than technical concerns such as data access or messaging.
   - An aggregate should have high functional cohesion.
   - An aggregate is a boundary of persistence.
   - Aggregates should be loosely coupled.

3. Domain services are also good candidates for microservices. Domain services are stateless operations

across multiple aggregates. A typical example is a workflow that involves several microservices. We'll see an example of this in the Drone Delivery application.

4. Finally, consider non-functional requirements. Look at factors such as team size, data types, technologies, scalability requirements, availability requirements, and security requirements. These factors may lead you to further decompose a microservice into two or more smaller services, or do the opposite and combine several microservices into one.

After you identify the microservices in your application, validate your design against the following criteria:

- Each service has a single responsibility.
- There are no chatty calls between services. If splitting functionality into two services causes them to be overly chatty, it may be a symptom that these functions belong in the same service.
- Each service is small enough that it can be built by a small team working independently.
- There are no inter-dependencies that will require two or more services to be deployed in lock-step. It should always be possible to deploy a service without redeploying any other services.
- Services are not tightly coupled, and can evolve independently.
- Your service boundaries will not create problems with data consistency or integrity. Sometimes it's important to maintain data consistency by putting functionality into a single microservice. That said, consider whether you really need strong consistency. There are strategies for addressing eventual consistency in a distributed system, and the benefits of decomposing services often outweigh the challenges of managing eventual consistency.

Above all, it's important to be pragmatic, and remember that domain-driven design is an iterative process. When in doubt, start with more coarse-grained microservices. Splitting a microservice into two smaller services is easier than refactoring functionality across several existing microservices.

Example: Defining microservices for the Drone Delivery application

Recall that the development team had identified the four aggregates — Delivery, Package, Drone, and Account — and two domain services, Scheduler and Supervisor.

Delivery and Package are obvious candidates for microservices. The Scheduler and Supervisor coordinate the activities performed by other microservices, so it makes sense to implement these domain services as microservices.

Drone and Account are interesting because they belong to other bounded contexts. One option is for the Scheduler to call the Drone and Account bounded contexts directly. Another option is to create Drone and Account microservices inside the Shipping bounded context. These microservices would mediate between the bounded contexts, by exposing APIs or data schemas that are more suited to the Shipping context.

The details of the Drone and Account bounded contexts are beyond the scope of this guidance, so we created mock services for them in our reference implementation. But here are some factors to consider in this situation:

- What is the network overhead of calling directly into the other bounded context?
- Is the data schema for the other bounded context suitable for this context, or is it better to have a schema that's tailored to this bounded context?
- Is the other bounded context a legacy system? If so, you might create a service that acts as an anti-corruption layer to translate between the legacy system and the modern application.
- What is the team structure? Is it easy to communicate with the team that's responsible for the other bounded context? If not, creating a service that mediates between the two contexts can help to mitigate the cost of cross-team communication.

So far, we haven't considered any non-functional requirements. Thinking about the application's throughput

requirements, the development team decided to create a separate Ingestion microservice that is responsible for ingesting client requests. This microservice will implement load leveling by putting incoming requests into a buffer for processing. The Scheduler will read the requests from the buffer and execute the workflow.

Non-functional requirements led the team to create one additional service. All of the services so far have been about the process of scheduling and delivering packages in real time. But the system also needs to store the history of every delivery in long-term storage for data analysis. The team considered making this the responsibility of the Delivery service. However, the data storage requirements are quite different for historical analysis versus in-flight operations (see Data considerations). Therefore, the team decided to create a separate Delivery History service, which will listen for DeliveryTracking events from the Delivery service and write the events into long-term storage.

The following diagram shows the design at this point:

Next steps

At this point, you should have a clear understanding of the purpose and functionality of each microservice in your design. Now you can architect the system.

Design a microservices architecture

Design a microservices architecture

11/19/2021 • 2 minutes to read • Edit Online

Microservices have become a popular architectural style for building cloud applications that are resilient, highly scalable, independently deployable, and able to evolve quickly. To be more than just a buzzword, however, microservices require a different approach to designing and building applications.

In this set of articles, we explore how to build a microservices architecture on Azure. Topics include:

- Compute options for microservices
- Interservice communication
- API design
- API gateways
- Data considerations
- Design patterns

Prerequisites

Before reading these articles, you might start with the following:

- Introduction to microservices architectures. Understand the benefits and challenges of microservices, and when to use this style of architecture.
- Using domain analysis to model microservices. Learn a domain-driven approach to modeling microservices.

Reference implementation

To illustrate best practices for a microservices architecture, we created a reference implementation that we call the Drone Delivery application. This implementation runs on Kubernetes using Azure Kubernetes Service (AKS). You can find the reference implementation on GitHub.

Scenario

Fabrikam, Inc. is starting a drone delivery service. The company manages a fleet of drone aircraft. Businesses register with the service, and users can request a drone to pick up goods for delivery. When a customer schedules a pickup, a backend system assigns a drone and notifies the user with an estimated delivery time. While the delivery is in progress, the customer can track the location of the drone, with a continuously updated ETA.

This scenario involves a fairly complicated domain. Some of the business concerns include scheduling drones, tracking packages, managing user accounts, and storing and analyzing historical data. Moreover, Fabrikam wants to get to market quickly and then iterate quickly, adding new functionality and capabilities. The application needs to operate at cloud scale, with a high service level objective (SLO). Fabrikam also expects that different parts of the system will have very different requirements for data storage and querying. All of these considerations lead Fabrikam to choose a microservices architecture for the Drone Delivery application.

> **NOTE**
>
> For help in choosing between a microservices architecture and other architectural styles, see the Azure Application Architecture Guide.

Our reference implementation uses Kubernetes with Azure Kubernetes Service (AKS). However, many of the high-level architectural decisions and challenges will apply to any container orchestrator, including Azure Service Fabric.

Next steps

Choose a compute option

Choose an Azure compute option for microservices

11/19/2021 • 4 minutes to read • Edit Online

The term *compute* refers to the hosting model for the computing resources that your application runs on. For a microservices architecture, two approaches are especially popular:

- A service orchestrator that manages services running on dedicated nodes (VMs).
- A serverless architecture using functions as a service (FaaS).

While these aren't the only options, they are both proven approaches to building microservices. An application might include both approaches.

Service orchestrators

An orchestrator handles tasks related to deploying and managing a set of services. These tasks include placing services on nodes, monitoring the health of services, restarting unhealthy services, load balancing network traffic across service instances, service discovery, scaling the number of instances of a service, and applying configuration updates. Popular orchestrators include Kubernetes, Service Fabric, DC/OS, and Docker Swarm.

On the Azure platform, consider the following options:

- Azure Kubernetes Service (AKS) is a managed Kubernetes service. AKS provisions Kubernetes and exposes the Kubernetes API endpoints, but hosts and manages the Kubernetes control plane, performing automated upgrades, automated patching, autoscaling, and other management tasks. You can think of AKS as being "Kubernetes APIs as a service."

- Service Fabric is a distributed systems platform for packaging, deploying, and managing microservices. Microservices can be deployed to Service Fabric as containers, as binary executables, or as Reliable Services. Using the Reliable Services programming model, services can directly use Service Fabric programming APIs to query the system, report health, receive notifications about configuration and code changes, and discover other services. A key differentiation with Service Fabric is its strong focus on building stateful services using Reliable Collections.

- Other options such as Docker Enterprise Edition and Mesosphere DC/OS can run in an IaaS environment on Azure. You can find deployment templates on Azure Marketplace.

Containers

Sometimes people talk about containers and microservices as if they were the same thing. While that's not true — you don't need containers to build microservices — containers do have some benefits that are particularly relevant to microservices, such as:

- **Portability**. A container image is a standalone package that runs without needing to install libraries or other dependencies. That makes them easy to deploy. Containers can be started and stopped quickly, so you can spin up new instances to handle more load or to recover from node failures.

- **Density**. Containers are lightweight compared with running a virtual machine, because they share OS resources. That makes it possible to pack multiple containers onto a single node, which is especially useful when the application consists of many small services.

- **Resource isolation**. You can limit the amount of memory and CPU that is available to a container, which can help to ensure that a runaway process doesn't exhaust the host resources. See the Bulkhead pattern for more information.

Serverless (Functions as a Service)

With a serverless architecture, you don't manage the VMs or the virtual network infrastructure. Instead, you deploy code and the hosting service handles putting that code onto a VM and executing it. This approach tends to favor small granular functions that are coordinated using event-based triggers. For example, a message being placed onto a queue might trigger a function that reads from the queue and processes the message.

Azure Functions is a serverless compute service that supports various function triggers, including HTTP requests, Service Bus queues, and Event Hubs events. For a complete list, see Azure Functions triggers and bindings concepts. Also consider Azure Event Grid, which is a managed event routing service in Azure.

Orchestrator or serverless?

Here are some factors to consider when choosing between an orchestrator approach and a serverless approach.

Manageability A serverless application is easy to manage, because the platform manages all the of compute resources for you. While an orchestrator abstracts some aspects of managing and configuring a cluster, it does not completely hide the underlying VMs. With an orchestrator, you will need to think about issues such as load balancing, CPU and memory usage, and networking.

Flexibility and control. An orchestrator gives you a great deal of control over configuring and managing your services and the cluster. The tradeoff is additional complexity. With a serverless architecture, you give up some degree of control because these details are abstracted.

Portability. All of the orchestrators listed here (Kubernetes, DC/OS, Docker Swarm, and Service Fabric) can run on-premises or in multiple public clouds.

Application integration. It can be challenging to build a complex application using a serverless architecture, due to the need to coordinate, deploy, and manage many small independent functions. One option in Azure is to use Azure Logic Apps to coordinate a set of Azure Functions. For an example of this approach, see Create a function that integrates with Azure Logic Apps.

Cost. With an orchestrator, you pay for the VMs that are running in the cluster. With a serverless application, you pay only for the actual compute resources consumed. In both cases, you need to factor in the cost of any additional services, such as storage, databases, and messaging services.

Scalability. Azure Functions scales automatically to meet demand, based on the number of incoming events. With an orchestrator, you can scale out by increasing the number of service instances running in the cluster. You can also scale by adding additional VMs to the cluster.

Our reference implementation primarily uses Kubernetes, but we did use Azure Functions for one service, namely the Delivery History service. Azure Functions was a good fit for this particular service, because it's is an event-driven workload. By using an Event Hubs trigger to invoke the function, the service needed a minimal amount of code. Also, the Delivery History service is not part of the main workflow, so running it outside of the Kubernetes cluster doesn't affect the end-to-end latency of user-initiated operations.

Next steps

Interservice communication

Designing interservice communication for microservices

11/19/2021 • 12 minutes to read • Edit Online

Communication between microservices must be efficient and robust. With lots of small services interacting to complete a single business activity, this can be a challenge. In this article, we look at the tradeoffs between asynchronous messaging versus synchronous APIs. Then we look at some of the challenges in designing resilient interservice communication.

Challenges

Here are some of the main challenges arising from service-to-service communication. Service meshes, described later in this article, are designed to handle many of these challenges.

Resiliency. There may be dozens or even hundreds of instances of any given microservice. An instance can fail for any number of reasons. There can be a node-level failure, such as a hardware failure or a VM reboot. An instance might crash, or be overwhelmed with requests and unable to process any new requests. Any of these events can cause a network call to fail. There are two design patterns that can help make service-to-service network calls more resilient:

- **Retry**. A network call may fail because of a transient fault that goes away by itself. Rather than fail outright, the caller should typically retry the operation a certain number of times, or until a configured time-out period elapses. However, if an operation is not idempotent, retries can cause unintended side effects. The original call might succeed, but the caller never gets a response. If the caller retries, the operation may be invoked twice. Generally, it's not safe to retry POST or PATCH methods, because these are not guaranteed to be idempotent.
- **Circuit Breaker**. Too many failed requests can cause a bottleneck, as pending requests accumulate in the queue. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on, which can cause cascading failures. The Circuit Breaker pattern can prevent a service from repeatedly trying an operation that is likely to fail.

Load balancing. When service "A" calls service "B", the request must reach a running instance of service "B". In Kubernetes, the `Service` resource type provides a stable IP address for a group of pods. Network traffic to the service's IP address gets forwarded to a pod by means of iptable rules. By default, a random pod is chosen. A service mesh (see below) can provide more intelligent load balancing algorithms based on observed latency or other metrics.

Distributed tracing. A single transaction may span multiple services. That can make it hard to monitor the overall performance and health of the system. Even if every service generates logs and metrics, without some way to tie them together, they are of limited use. The article Logging and monitoring talks more about distributed tracing, but we mention it here as a challenge.

Service versioning. When a team deploys a new version of a service, they must avoid breaking any other services or external clients that depend on it. In addition, you might want to run multiple versions of a service side-by-side, and route requests to a particular version. See API Versioning for more discussion of this issue.

TLS encryption and mutual TLS authentication. For security reasons, you may want to encrypt traffic between services with TLS, and use mutual TLS authentication to authenticate callers.

Synchronous versus asynchronous messaging

There are two basic messaging patterns that microservices can use to communicate with other microservices.

1. Synchronous communication. In this pattern, a service calls an API that another service exposes, using a protocol such as HTTP or gRPC. This option is a synchronous messaging pattern because the caller waits for a response from the receiver.
2. Asynchronous message passing. In this pattern, a service sends message without waiting for a response, and one or more services process the message asynchronously.

It's important to distinguish between asynchronous I/O and an asynchronous protocol. Asynchronous I/O means the calling thread is not blocked while the I/O completes. That's important for performance, but is an implementation detail in terms of the architecture. An asynchronous protocol means the sender doesn't wait for a response. HTTP is a synchronous protocol, even though an HTTP client may use asynchronous I/O when it sends a request.

There are tradeoffs to each pattern. Request/response is a well-understood paradigm, so designing an API may feel more natural than designing a messaging system. However, asynchronous messaging has some advantages that can be useful in a microservices architecture:

- **Reduced coupling**. The message sender does not need to know about the consumer.
- **Multiple subscribers**. Using a pub/sub model, multiple consumers can subscribe to receive events. See Event-driven architecture style.
- **Failure isolation**. If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers. This ability is especially useful in a microservices architecture, because each service has its own lifecycle. A service could become unavailable or be replaced with a newer version at any given time. Asynchronous messaging can handle intermittent downtime. Synchronous APIs, on the other hand, require the downstream service to be available or the operation fails.
- **Responsiveness**. An upstream service can reply faster if it does not wait on downstream services. This is especially useful in a microservices architecture. If there is a chain of service dependencies (service A calls B, which calls C, and so on), waiting on synchronous calls can add unacceptable amounts of latency.
- **Load leveling**. A queue can act as a buffer to level the workload, so that receivers can process messages at their own rate.
- **Workflows**. Queues can be used to manage a workflow, by check-pointing the message after each step in the workflow.

However, there are also some challenges to using asynchronous messaging effectively.

- **Coupling with the messaging infrastructure**. Using a particular messaging infrastructure may cause tight coupling with that infrastructure. It will be difficult to switch to another messaging infrastructure later.
- **Latency**. End-to-end latency for an operation may become high if the message queues fill up.
- **Cost**. At high throughputs, the monetary cost of the messaging infrastructure could be significant.
- **Complexity**. Handling asynchronous messaging is not a trivial task. For example, you must handle duplicated messages, either by de-duplicating or by making operations idempotent. It's also hard to implement request-response semantics using asynchronous messaging. To send a response, you need another queue, plus a way to correlate request and response messages.
- **Throughput**. If messages require *queue semantics*, the queue can become a bottleneck in the system. Each message requires at least one queue operation and one dequeue operation. Moreover, queue

semantics generally require some kind of locking inside the messaging infrastructure. If the queue is a managed service, there may be additional latency, because the queue is external to the cluster's virtual network. You can mitigate these issues by batching messages, but that complicates the code. If the messages don't require queue semantics, you might be able to use an event *stream* instead of a queue. For more information, see Event-driven architectural style.

Drone Delivery: Choosing the messaging patterns

With these considerations in mind, the development team made the following design choices for the Drone Delivery application

- The Ingestion service exposes a public REST API that client applications use to schedule, update, or cancel deliveries.
- The Ingestion service uses Event Hubs to send asynchronous messages to the Scheduler service. Asynchronous messages are necessary to implement the load-leveling that is required for ingestion.
- The Account, Delivery, Package, Drone, and Third-party Transport services all expose internal REST APIs. The Scheduler service calls these APIs to carry out a user request. One reason to use synchronous APIs is that the Scheduler needs to get a response from each of the downstream services. A failure in any of the downstream services means the entire operation failed. However, a potential issue is the amount of latency that is introduced by calling the backend services.
- If any downstream service has a nontransient failure, the entire transaction should be marked as failed. To handle this case, the Scheduler service sends an asynchronous message to the Supervisor, so that the Supervisor can schedule compensating transactions.
- The Delivery service exposes a public API that clients can use to get the status of a delivery. In the article API gateway, we discuss how an API gateway can hide the underlying services from the client, so the client doesn't need to know which services expose which APIs.
- While a drone is in flight, the Drone service sends events that contain the drone's current location and status. The Delivery service listens to these events in order to track the status of a delivery.
- When the status of a delivery changes, the Delivery service sends a delivery status event, such as `DeliveryCreated` or `DeliveryCompleted`. Any service can subscribe to these events. In the current design, the Delivery History service is the only subscriber, but there might be other subscribers later. For example, the events might go to a real-time analytics service. And because the Scheduler doesn't have to wait for a response, adding more subscribers doesn't affect the main workflow path.

Notice that delivery status events are derived from drone location events. For example, when a drone reaches a delivery location and drops off a package, the Delivery service translates this into a DeliveryCompleted event. This is an example of thinking in terms of domain models. As described earlier, Drone Management belongs in a separate bounded context. The drone events convey the physical location of a drone. The delivery events, on the other hand, represent changes in the status of a delivery, which is a different business entity.

Using a service mesh

A *service mesh* is a software layer that handles service-to-service communication. Service meshes are designed to address many of the concerns listed in the previous section, and to move responsibility for these concerns away from the microservices themselves and into a shared layer. The service mesh acts as a proxy that intercepts network communication between microservices in the cluster. Currently, the service mesh concept applies mainly to container orchestrators, rather than serverless architectures.

> **NOTE**
> Service mesh is an example of the Ambassador pattern — a helper service that sends network requests on behalf of the application.

Right now, the main options for a service mesh in Kubernetes are linkerd and Istio. Both of these technologies are evolving rapidly. However, some features that both linkerd and Istio have in common include:

- Load balancing at the session level, based on observed latencies or number of outstanding requests. This can improve performance over the layer-4 load balancing that is provided by Kubernetes.
- Layer-7 routing based on URL path, Host header, API version, or other application-level rules.
- Retry of failed requests. A service mesh understands HTTP error codes, and can automatically retry failed requests. You can configure that maximum number of retries, along with a timeout period in order to bound the maximum latency.
- Circuit breaking. If an instance consistently fails requests, the service mesh will temporarily mark it as unavailable. After a backoff period, it will try the instance again. You can configure the circuit breaker based on various criteria, such as the number of consecutive failures,
- Service mesh captures metrics about interservice calls, such as the request volume, latency, error and

success rates, and response sizes. The service mesh also enables distributed tracing by adding correlation information for each hop in a request.

- Mutual TLS Authentication for service-to-service calls.

Do you need a service mesh? It depends. Without a service mesh, you'll need to consider each of the challenges mentioned at the beginning of this article. You can solve problems like retry, circuit breaker, and distributed tracing without a service mesh, but a service mesh moves these concerns out of the individual services and into a dedicated layer. On the other hand, a service mesh adds complexity to the setup and configuration of the cluster. There may be performance implications, because requests now get routed through the service mesh proxy, and because extra services are now running on every node in the cluster. You should do thorough performance and load testing before deploying a service mesh in production.

Distributed transactions

A common challenge in microservices is correctly handling transactions that span multiple services. Often in this scenario, the success of a transaction is all or nothing — if one of the participating services fails, the entire transaction must fail.

There are two cases to consider:

- A service may experience a *transient* failure such as a network timeout. These errors can often be resolved simply by retrying the call. If the operation still fails after a certain number of attempts, it's considered a nontransient failure.
- A *nontransient* failure is any failure that's unlikely to go away by itself. Nontransient failures include normal error conditions, such as invalid input. They also include unhandled exceptions in application code or a process crashing. If this type of error occurs, the entire business transaction must be marked as a failure. It may be necessary to undo other steps in the same transaction that already succeeded.

After a nontransient failure, the current transaction might be in a *partially failed* state, where one or more steps already completed successfully. For example, if the Drone service already scheduled a drone, the drone must be canceled. In that case, the application needs to undo the steps that succeeded, by using a Compensating Transaction. In some cases, this must be done by an external system or even by a manual process.

If the logic for compensating transactions is complex, consider creating a separate service that is responsible for this process. In the Drone Delivery application, the Scheduler service puts failed operations onto a dedicated queue. A separate microservice, called the Supervisor, reads from this queue and calls a cancellation API on the services that need to compensate. This is a variation of the Scheduler Agent Supervisor pattern. The Supervisor service might take other actions as well, such as notify the user by text or email, or send an alert to an operations dashboard.

The Scheduler service itself might fail (for example, because a node crashes). In that case, a new instance can spin up and take over. However, any transactions that were already in progress must be resumed.

One approach is to save a checkpoint to a durable store after each step in the workflow is completed. If an instance of the Scheduler service crashes in the middle of a transaction, a new instance can use the checkpoint to resume where the previous instance left off. However, writing checkpoints can create a performance overhead.

Another option is to design all operations to be idempotent. An operation is idempotent if it can be called multiple times without producing additional side-effects after the first call. Essentially, the downstream service should ignore duplicate calls, which means the service must be able to detect duplicate calls. It's not always straightforward to implement idempotent methods. For more information, see Idempotent operations.

Next steps

For microservices that talk directly to each other, it's important to create well-designed APIs.

API design

Designing APIs for microservices

11/19/2021 • 12 minutes to read • Edit Online

Good API design is important in a microservices architecture, because all data exchange between services happens either through messages or API calls. APIs must be efficient to avoid creating chatty I/O. Because services are designed by teams working independently, APIs must have well-defined semantics and versioning schemes, so that updates don't break other services.

It's important to distinguish between two types of API:

- Public APIs that client applications call.
- Backend APIs that are used for interservice communication.

These two use cases have somewhat different requirements. A public API must be compatible with client applications, typically browser applications or native mobile applications. Most of the time, that means the public API will use REST over HTTP. For the backend APIs, however, you need to take network performance into account. Depending on the granularity of your services, interservice communication can result in a lot of network traffic. Services can quickly become I/O bound. For that reason, considerations such as serialization speed and payload size become more important. Some popular alternatives to using REST over HTTP include gRPC, Apache Avro, and Apache Thrift. These protocols support binary serialization and are generally more efficient than HTTP.

Considerations

Here are some things to think about when choosing how to implement an API.

REST versus RPC. Consider the tradeoffs between using a REST-style interface versus an RPC-style interface.

- REST models resources, which can be a natural way to express your domain model. It defines a uniform interface based on HTTP verbs, which encourages evolvability. It has well-defined semantics in terms of idempotency, side effects, and response codes. And it enforces stateless communication, which improves scalability.
- RPC is more oriented around operations or commands. Because RPC interfaces look like local method calls, it may lead you to design overly chatty APIs. However, that doesn't mean RPC must be chatty. It just

means you need to use care when designing the interface.

For a RESTful interface, the most common choice is REST over HTTP using JSON. For an RPC-style interface, there are several popular frameworks, including gRPC, Apache Avro, and Apache Thrift.

Efficiency. Consider efficiency in terms of speed, memory, and payload size. Typically a gRPC-based interface is faster than REST over HTTP.

Interface definition language (IDL). An IDL is used to define the methods, parameters, and return values of an API. An IDL can be used to generate client code, serialization code, and API documentation. IDLs can also be consumed by API testing tools such as Postman. Frameworks such as gRPC, Avro, and Thrift define their own IDL specifications. REST over HTTP does not have a standard IDL format, but a common choice is OpenAPI (formerly Swagger). You can also create an HTTP REST API without using a formal definition language, but then you lose the benefits of code generation and testing.

Serialization. How are objects serialized over the wire? Options include text-based formats (primarily JSON) and binary formats such as protocol buffer. Binary formats are generally faster than text-based formats. However, JSON has advantages in terms of interoperability, because most languages and frameworks support JSON serialization. Some serialization formats require a fixed schema, and some require compiling a schema definition file. In that case, you'll need to incorporate this step into your build process.

Framework and language support. HTTP is supported in nearly every framework and language. gRPC, Avro, and Thrift all have libraries for C++, C#, Java, and Python. Thrift and gRPC also support Go.

Compatibility and interoperability. If you choose a protocol like gRPC, you may need a protocol translation layer between the public API and the back end. A gateway can perform that function. If you are using a service mesh, consider which protocols are compatible with the service mesh. For example, linkerd has built-in support for HTTP, Thrift, and gRPC.

Our baseline recommendation is to choose REST over HTTP unless you need the performance benefits of a binary protocol. REST over HTTP requires no special libraries. It creates minimal coupling, because callers don't need a client stub to communicate with the service. There are rich ecosystems of tools to support schema definitions, testing, and monitoring of RESTful HTTP endpoints. Finally, HTTP is compatible with browser clients, so you don't need a protocol translation layer between the client and the backend.

However, if you choose REST over HTTP, you should do performance and load testing early in the development process, to validate whether it performs well enough for your scenario.

RESTful API design

There are many resources for designing RESTful APIs. Here are some that you might find helpful:

- API design
- API implementation
- Microsoft REST API Guidelines

Here are some specific considerations to keep in mind.

- Watch out for APIs that leak internal implementation details or simply mirror an internal database schema. The API should model the domain. It's a contract between services, and ideally should only change when new functionality is added, not just because you refactored some code or normalized a database table.
- Different types of client, such as mobile application and desktop web browser, may require different payload sizes or interaction patterns. Consider using the Backends for Frontends pattern to create separate backends for each client, that expose an optimal interface for that client.

- For operations with side effects, consider making them idempotent and implementing them as PUT methods. That will enable safe retries and can improve resiliency. The article Interservice communication discuss this issue in more detail.
- HTTP methods can have asynchronous semantics, where the method returns a response immediately, but the service carries out the operation asynchronously. In that case, the method should return an HTTP 202 response code, which indicates the request was accepted for processing, but the processing is not yet completed. For more information, see Asynchronous Request-Reply pattern.

Mapping REST to DDD patterns

Patterns such as entity, aggregate, and value object are designed to place certain constraints on the objects in your domain model. In many discussions of DDD, the patterns are modeled using object-oriented (OO) language concepts like constructors or property getters and setters. For example, *value objects* are supposed to be immutable. In an OO programming language, you would enforce this by assigning the values in the constructor and making the properties read-only:

```
export class Location {
    readonly latitude: number;
    readonly longitude: number;

    constructor(latitude: number, longitude: number) {
        if (latitude < -90 || latitude > 90) {
            throw new RangeError('latitude must be between -90 and 90');
        }
        if (longitude < -180 || longitude > 180) {
            throw new RangeError('longitude must be between -180 and 180');
        }
        this.latitude = latitude;
        this.longitude = longitude;
    }
}
```

These sorts of coding practices are particularly important when building a traditional monolithic application. With a large code base, many subsystems might use the `Location` object, so it's important for the object to enforce correct behavior.

Another example is the Repository pattern, which ensures that other parts of the application do not make direct reads or writes to the data store:

In a microservices architecture, however, services don't share the same code base and don't share data stores. Instead, they communicate through APIs. Consider the case where the Scheduler service requests information about a drone from the Drone service. The Drone service has its internal model of a drone, expressed through code. But the Scheduler doesn't see that. Instead, it gets back a *representation* of the drone entity — perhaps a JSON object in an HTTP response.

The Scheduler service can't modify the Drone service's internal models, or write to the Drone service's data store. That means the code that implements the Drone service has a smaller exposed surface area, compared with code in a traditional monolith. If the Drone service defines a Location class, the scope of that class is limited — no other service will directly consume the class.

For these reasons, this guidance doesn't focus much on coding practices as they relate to the tactical DDD patterns. But it turns out that you can also model many of the DDD patterns through REST APIs.

For example:

- Aggregates map naturally to *resources* in REST. For example, the Delivery aggregate would be exposed as a resource by the Delivery API.
- Aggregates are consistency boundaries. Operations on aggregates should never leave an aggregate in an inconsistent state. Therefore, you should avoid creating APIs that allow a client to manipulate the internal state of an aggregate. Instead, favor coarse-grained APIs that expose aggregates as resources.
- Entities have unique identities. In REST, resources have unique identifiers in the form of URLs. Create resource URLs that correspond to an entity's domain identity. The mapping from URL to domain identity may be opaque to client.
- Child entities of an aggregate can be reached by navigating from the root entity. If you follow HATEOAS principles, child entities can be reached via links in the representation of the parent entity.
- Because value objects are immutable, updates are performed by replacing the entire value object. In REST, implement updates through PUT or PATCH requests.
- A repository lets clients query, add, or remove objects in a collection, abstracting the details of the underlying data store. In REST, a collection can be a distinct resource, with methods for querying the collection or adding new entities to the collection.

When you design your APIs, think about how they express the domain model, not just the data inside the model, but also the business operations and the constraints on the data.

DDD CONCEPT	REST EQUIVALENT	EXAMPLE
Aggregate	Resource	`{ "1":1234, "status":"pending"... }`
Identity	URL	`https://delivery-service/deliveries/1`
Child entities	Links	`{ "href": "/deliveries/1/confirmation" }`
Update value objects	PUT or PATCH	`PUT https://delivery-service/deliveries/1/dropoff`

DDD CONCEPT	REST EQUIVALENT	EXAMPLE
Repository	Collection	`https://delivery-service/deliveries?status=pending`

API versioning

An API is a contract between a service and clients or consumers of that service. If an API changes, there is a risk of breaking clients that depend on the API, whether those are external clients or other microservices. Therefore, it's a good idea to minimize the number of API changes that you make. Often, changes in the underlying implementation don't require any changes to the API. Realistically, however, at some point you will want to add new features or new capabilities that require changing an existing API.

Whenever possible, make API changes backward compatible. For example, avoid removing a field from a model, because that can break clients that expect the field to be there. Adding a field does not break compatibility, because clients should ignore any fields they don't understand in a response. However, the service must handle the case where an older client omits the new field in a request.

Support versioning in your API contract. If you introduce a breaking API change, introduce a new API version. Continue to support the previous version, and let clients select which version to call. There are a couple of ways to do this. One is simply to expose both versions in the same service. Another option is to run two versions of the service side-by-side, and route requests to one or the other version, based on HTTP routing rules.

The diagram has two parts. "Service supports two versions" shows the v1 Client and the v2 Client both pointing to one Service. "Side-by-side deployment" shows the v1 Client pointing to a v1 Service, and the v2 Client pointing to a v2 Service.

There's a cost to supporting multiple versions, in terms of developer time, testing, and operational overhead. Therefore, it's good to deprecate old versions as quickly as possible. For internal APIs, the team that owns the API can work with other teams to help them migrate to the new version. This is when having a cross-team governance process is useful. For external (public) APIs, it can be harder to deprecate an API version, especially if the API is consumed by third parties or by native client applications.

When a service implementation changes, it's useful to tag the change with a version. The version provides important information when troubleshooting errors. It can be very helpful for root cause analysis to know exactly which version of the service was called. Consider using semantic versioning for service versions. Semantic versioning uses a *MAJOR.MINOR.PATCH* format. However, clients should only select an API by the

major version number, or possibly the minor version if there are significant (but non-breaking) changes between minor versions. In other words, it's reasonable for clients to select between version 1 and version 2 of an API, but not to select version 2.1.3. If you allow that level of granularity, you risk having to support a proliferation of versions.

For further discussion of API versioning, see Versioning a RESTful web API.

Idempotent operations

An operation is *idempotent* if it can be called multiple times without producing additional side-effects after the first call. Idempotency can be a useful resiliency strategy, because it allows an upstream service to safely invoke an operation multiple times. For a discussion of this point, see Distributed transactions.

The HTTP specification states that GET, PUT, and DELETE methods must be idempotent. POST methods are not guaranteed to be idempotent. If a POST method creates a new resource, there is generally no guarantee that this operation is idempotent. The specification defines idempotent this way:

> A request method is considered "idempotent" if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request. (RFC 7231)

It's important to understand the difference between PUT and POST semantics when creating a new entity. In both cases, the client sends a representation of an entity in the request body. But the meaning of the URI is different.

- For a POST method, the URI represents a parent resource of the new entity, such as a collection. For example, to create a new delivery, the URI might be `/api/deliveries`. The server creates the entity and assigns it a new URI, such as `/api/deliveries/39660`. This URI is returned in the Location header of the response. Each time the client sends a request, the server will create a new entity with a new URI.
- For a PUT method, the URI identifies the entity. If there already exists an entity with that URI, the server replaces the existing entity with the version in the request. If no entity exists with that URI, the server creates one. For example, suppose the client sends a PUT request to `api/deliveries/39660`. Assuming there is no delivery with that URI, the server creates a new one. Now if the client sends the same request again, the server will replace the existing entity.

Here is the Delivery service's implementation of the PUT method.

```
[HttpPut("{id}")]
[ProducesResponseType(typeof(Delivery), 201)]
[ProducesResponseType(typeof(void), 204)]
public async Task<IActionResult> Put([FromBody]Delivery delivery, string id)
{
    logger.LogInformation("In Put action with delivery {Id}: {@DeliveryInfo}", id, delivery.ToLogInfo());
    try
    {
        var internalDelivery = delivery.ToInternal();

        // Create the new delivery entity.
        await deliveryRepository.CreateAsync(internalDelivery);

        // Create a delivery status event.
        var deliveryStatusEvent = new DeliveryStatusEvent { DeliveryId = delivery.Id, Stage =
DeliveryEventType.Created };
        await deliveryStatusEventRepository.AddAsync(deliveryStatusEvent);

        // Return HTTP 201 (Created)
        return CreatedAtRoute("GetDelivery", new { id= delivery.Id }, delivery);
    }
    catch (DuplicateResourceException)
    {
        // This method is mainly used to create deliveries. If the delivery already exists then update it.
        logger.LogInformation("Updating resource with delivery id: {DeliveryId}", id);

        var internalDelivery = delivery.ToInternal();
        await deliveryRepository.UpdateAsync(id, internalDelivery);

        // Return HTTP 204 (No Content)
        return NoContent();
    }
}
```

It's expected that most requests will create a new entity, so the method optimistically calls `CreateAsync` on the repository object, and then handles any duplicate-resource exceptions by updating the resource instead.

Next steps

Learn about using an API gateway at the boundary between client applications and microservices.

API gateways

Using API gateways in microservices

11/19/2021 • 5 minutes to read • Edit Online

In a microservices architecture, a client might interact with more than one front-end service. Given this fact, how does a client know what endpoints to call? What happens when new services are introduced, or existing services are refactored? How do services handle SSL termination, authentication, and other concerns? An *API gateway* can help to address these challenges.

What is an API gateway?

An API gateway sits between clients and services. It acts as a reverse proxy, routing requests from clients to services. It may also perform various cross-cutting tasks such as authentication, SSL termination, and rate limiting. If you don't deploy a gateway, clients must send requests directly to front-end services. However, there are some potential problems with exposing services directly to clients:

- It can result in complex client code. The client must keep track of multiple endpoints, and handle failures in a resilient way.
- It creates coupling between the client and the backend. The client needs to know how the individual services are decomposed. That makes it harder to maintain the client and also harder to refactor services.
- A single operation might require calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency.
- Each public-facing service must handle concerns such as authentication, SSL, and client rate limiting.
- Services must expose a client-friendly protocol such as HTTP or WebSocket. This limits the choice of communication protocols.
- Services with public endpoints are a potential attack surface, and must be hardened.

A gateway helps to address these issues by decoupling clients from services. Gateways can perform a number of different functions, and you may not need all of them. The functions can be grouped into the following design patterns:

Gateway Routing. Use the gateway as a reverse proxy to route requests to one or more backend services, using layer 7 routing. The gateway provides a single endpoint for clients, and helps to decouple clients from services.

Gateway Aggregation. Use the gateway to aggregate multiple individual requests into a single request. This pattern applies when a single operation requires calls to multiple backend services. The client sends one request to the gateway. The gateway dispatches requests to the various backend services, and then aggregates the results and sends them back to the client. This helps to reduce chattiness between the client and the backend.

Gateway Offloading. Use the gateway to offload functionality from individual services to the gateway, particularly cross-cutting concerns. It can be useful to consolidate these functions into one place, rather than making every service responsible for implementing them. This is particularly true for features that requires specialized skills to implement correctly, such as authentication and authorization.

Here are some examples of functionality that could be offloaded to a gateway:

- SSL termination
- Authentication
- IP allow/block list
- Client rate limiting (throttling)
- Logging and monitoring
- Response caching
- Web application firewall
- GZIP compression
- Servicing static content

Choosing a gateway technology

Here are some options for implementing an API gateway in your application.

- **Reverse proxy server**. Nginx and HAProxy are popular reverse proxy servers that support features such as load balancing, SSL, and layer 7 routing. They are both free, open-source products, with paid editions that provide additional features and support options. Nginx and HAProxy are both mature products with rich feature sets and high performance. You can extend them with third-party modules or by writing custom scripts in Lua. Nginx also supports a JavaScript-based scripting module referred to as NGINX JavaScript. This module was formally named nginScript.

- **Service mesh ingress controller**. If you are using a service mesh such as linkerd or Istio, consider the features that are provided by the ingress controller for that service mesh. For example, the Istio ingress controller supports layer 7 routing, HTTP redirects, retries, and other features.

- Azure Application Gateway. Application Gateway is a managed load balancing service that can perform layer-7 routing and SSL termination. It also provides a web application firewall (WAF).

- Azure API Management. API Management is a turnkey solution for publishing APIs to external and internal customers. It provides features that are useful for managing a public-facing API, including rate limiting, IP restrictions, and authentication using Azure Active Directory or other identity providers. API Management doesn't perform any load balancing, so it should be used in conjunction with a load balancer such as Application Gateway or a reverse proxy. For information about using API Management with Application Gateway, see Integrate API Management in an internal VNet with Application Gateway.

When choosing a gateway technology, consider the following:

Features. The options listed above all support layer 7 routing, but support for other features will vary. Depending on the features that you need, you might deploy more than one gateway.

Deployment. Azure Application Gateway and API Management are managed services. Nginx and HAProxy will typically run in containers inside the cluster, but can also be deployed to dedicated VMs outside of the cluster. This isolates the gateway from the rest of the workload, but incurs higher management overhead.

Management. When services are updated or new services are added, the gateway routing rules may need to be updated. Consider how this process will be managed. Similar considerations apply to managing SSL certificates, IP allow lists, and other aspects of configuration.

Deploying Nginx or HAProxy to Kubernetes

You can deploy Nginx or HAProxy to Kubernetes as a ReplicaSet or DaemonSet that specifies the Nginx or HAProxy container image. Use a ConfigMap to store the configuration file for the proxy, and mount the ConfigMap as a volume. Create a service of type LoadBalancer to expose the gateway through an Azure Load Balancer.

An alternative is to create an Ingress Controller. An Ingress Controller is a Kubernetes resource that deploys a load balancer or reverse proxy server. Several implementations exist, including Nginx and HAProxy. A separate resource called an Ingress defines settings for the Ingress Controller, such as routing rules and TLS certificates. That way, you don't need to manage complex configuration files that are specific to a particular proxy server technology.

The gateway is a potential bottleneck or single point of failure in the system, so always deploy at least two replicas for high availability. You may need to scale out the replicas further, depending on the load.

Also consider running the gateway on a dedicated set of nodes in the cluster. Benefits to this approach include:

- Isolation. All inbound traffic goes to a fixed set of nodes, which can be isolated from backend services.
- Stable configuration. If the gateway is misconfigured, the entire application may become unavailable.
- Performance. You may want to use a specific VM configuration for the gateway for performance reasons.

Next steps

The previous articles have looked at the interfaces *between* microservices or between microservices and client applications. By design, these interfaces treat each service as a opaque box. In particular, microservices should never expose implementation details about how they manage data. That has implications for data integrity and data consistency, explored in the next article.

Data considerations for microservices

Data considerations for microservices

11/19/2021 • 7 minutes to read • Edit Online

This article describes considerations for managing data in a microservices architecture. Because every microservice manages its own data, data integrity and data consistency are critical challenges.

A basic principle of microservices is that each service manages its own data. Two services should not share a data store. Instead, each service is responsible for its own private data store, which other services cannot access directly.

The reason for this rule is to avoid unintentional coupling between services, which can result if services share the same underlying data schemas. If there is a change to the data schema, the change must be coordinated across every service that relies on that database. By isolating each service's data store, we can limit the scope of change, and preserve the agility of truly independent deployments. Another reason is that each microservice may have its own data models, queries, or read/write patterns. Using a shared data store limits each team's ability to optimize data storage for their particular service.

This approach naturally leads to polyglot persistence — the use of multiple data storage technologies within a single application. One service might require the schema-on-read capabilities of a document database. Another might need the referential integrity provided by an RDBMS. Each team is free to make the best choice for their service. For more about the general principle of polyglot persistence, see Use the best data store for the job.

> **NOTE**
> It's fine for services to share the same physical database server. The problem occurs when services share the same schema, or read and write to the same set of database tables.

Challenges

Some challenges arise from this distributed approach to managing data. First, there may be redundancy across the data stores, with the same item of data appearing in multiple places. For example, data might be stored as part of a transaction, then stored elsewhere for analytics, reporting, or archiving. Duplicated or partitioned data can lead to issues of data integrity and consistency. When data relationships span multiple services, you can't use traditional data management techniques to enforce the relationships.

Traditional data modeling uses the rule of "one fact in one place." Every entity appears exactly once in the schema. Other entities may hold references to it but not duplicate it. The obvious advantage to the traditional approach is that updates are made in a single place, which avoids problems with data consistency. In a microservices architecture, you have to consider how updates are propagated across services, and how to manage eventual consistency when data appears in multiple places without strong consistency.

Approaches to managing data

There is no single approach that's correct in all cases, but here are some general guidelines for managing data in a microservices architecture.

- Embrace eventual consistency where possible. Understand the places in the system where you need strong consistency or ACID transactions, and the places where eventual consistency is acceptable.
- When you need strong consistency guarantees, one service may represent the source of truth for a given entity, which is exposed through an API. Other services might hold their own copy of the data, or a subset of the data, that is eventually consistent with the master data but not considered the source of truth. For example, imagine an e-commerce system with a customer order service and a recommendation service. The recommendation service might listen to events from the order service, but if a customer requests a refund, it is the order service, not the recommendation service, that has the complete transaction history.
- For transactions, use patterns such as Scheduler Agent Supervisor and Compensating Transaction to keep data consistent across several services. You may need to store an additional piece of data that captures the state of a unit of work that spans multiple services, to avoid partial failure among multiple services. For example, keep a work item on a durable queue while a multi-step transaction is in progress.
- Store only the data that a service needs. A service might only need a subset of information about a domain entity. For example, in the Shipping bounded context, we need to know which customer is associated to a particular delivery. But we don't need the customer's billing address — that's managed by the Accounts bounded context. Thinking carefully about the domain, and using a DDD approach, can help here.
- Consider whether your services are coherent and loosely coupled. If two services are continually exchanging information with each other, resulting in chatty APIs, you may need to redraw your service boundaries, by merging two services or refactoring their functionality.
- Use an event driven architecture style. In this architecture style, a service publishes an event when there are changes to its public models or entities. Interested services can subscribe to these events. For example, another service could use the events to construct a materialized view of the data that is more suitable for querying.
- A service that owns events should publish a schema that can be used to automate serializing and deserializing the events, to avoid tight coupling between publishers and subscribers. Consider JSON schema or a framework like Microsoft Bond, Protobuf, or Avro.
- At high scale, events can become a bottleneck on the system, so consider using aggregation or batching to reduce the total load.

Example: Choosing data stores for the Drone Delivery application

The previous articles in this series discuss a drone delivery service as a running example. You can read more about the scenario and the corresponding reference implementation here.

To recap, this application defines several microservices for scheduling deliveries by drone. When a user schedules a new delivery, the client request includes information about the delivery, such as pickup and dropoff locations, and about the package, such as size and weight. This information defines a unit of work.

The various backend services care about different portions of the information in the request, and also have different read and write profiles.

Delivery service

The Delivery service stores information about every delivery that is currently scheduled or in progress. It listens for events from the drones, and tracks the status of deliveries that are in progress. It also sends domain events with delivery status updates.

It's expected that users will frequently check the status of a delivery while they are waiting for their package. Therefore, the Delivery service requires a data store that emphasizes throughput (read and write) over long-term storage. Also, the Delivery service does not perform any complex queries or analysis, it simply fetches the latest status for a given delivery. The Delivery service team chose Azure Cache for Redis for its high read-write performance. The information stored in Redis is relatively short-lived. Once a delivery is complete, the Delivery History service is the system of record.

Delivery History service

The Delivery History service listens for delivery status events from the Delivery service. It stores this data in long-term storage. There are two different use-cases for this historical data, which have different data storage requirements.

The first scenario is aggregating the data for the purpose of data analytics, in order to optimize the business or improve the quality of the service. Note that the Delivery History service doesn't perform the actual analysis of the data. It's only responsible for the ingestion and storage. For this scenario, the storage must be optimized for data analysis over a large set of data, using a schema-on-read approach to accommodate a variety of data sources. Azure Data Lake Store is a good fit for this scenario. Data Lake Store is an Apache Hadoop file system compatible with Hadoop Distributed File System (HDFS), and is tuned for performance for data analytics scenarios.

The other scenario is enabling users to look up the history of a delivery after the delivery is completed. Azure Data Lake is not optimized for this scenario. For optimal performance, Microsoft recommends storing time-series data in Data Lake in folders partitioned by date. (See Tuning Azure Data Lake Store for performance). However, that structure is not optimal for looking up individual records by ID. Unless you also know the timestamp, a lookup by ID requires scanning the entire collection. Therefore, the Delivery History service also stores a subset of the historical data in Cosmos DB for quicker lookup. The records don't need to stay in Cosmos DB indefinitely. Older deliveries can be archived — say, after a month. This could be done by running an occasional batch process.

Package service

The Package service stores information about all of the packages. The storage requirements for the Package are:

- Long-term storage.
- Able to handle a high volume of packages, requiring high write throughput.
- Support simple queries by package ID. No complex joins or requirements for referential integrity.

Because the package data is not relational, a document-oriented database is appropriate, and Cosmos DB can achieve high throughput by using sharded collections. The team that works on the Package service is familiar with the MEAN stack (MongoDB, Express.js, AngularJS, and Node.js), so they select the MongoDB API for Cosmos DB. That lets them leverage their existing experience with MongoDB, while getting the benefits of Cosmos DB, which is a managed Azure service.

Next steps

Learn about design patterns that can help mitigate some common challenges in a microservices architecture.

Design patterns for microservices

Container orchestration for microservices

11/19/2021 • 3 minutes to read • Edit Online

Microservices architectures typically package and deploy each microservice instance inside a single container. Many instances of the microservices might be running, each in a separate container. Containers are lightweight and short-lived, making them easy to create and destroy, but difficult to coordinate and communicate between.

This article discusses the challenges of running a containerized microservices architecture at production scale, and how container orchestration can help. The article presents several Azure container orchestration options.

Containerized microservices architecture

In this simple containerized Azure Kubernetes Service (AKS) cluster:

- One Microservice A instance is running in Node 1, another instance in Node 2, and a third instance in Node 3.
- One instance of Microservice B is running in Node 1, and another instance in Node 3.
- Containerized reverse proxy servers are running in Nodes 1 and 2 to distribute traffic.

Web app in a multiple-container AKS cluster with an orchestrator

To manage the cluster, a DevOps team has to:

- Run multiple container instances in each node.
- Load balance traffic between the instances.
- Manage communication between dependent instances in separate nodes.
- Maintain the desired AKS cluster state.

With container orchestration, the DevOps team can represent the cluster's desired state as a configuration. A container orchestration engine enforces the desired configuration and automates all the management tasks.

Consider containerizing a simple three-tier web application:

- A container hosts the front-end component.
- Another container hosts the middle tier or REST API layer.
- The middle tier layer communicates with a globally distributed database.

Running these containers on a single development machine might not be too hard. However, running the

application in high availability mode at scale in a production cluster quickly becomes challenging. Container orchestration is crucial for large and dynamic production environments.

Advantages of container orchestration

The following example shows how container orchestration can help manage cluster deployment, networking, and scaling.

The container orchestrator:

- Automatically scales the number of microservice instances, based on traffic or resource utilization. In the example, the orchestrator automatically adds another Microservice A instance in response to increased traffic.
- Manages the containers to reflect the configured desired state. In the example, Microservice B is configured to have two instances. One instance has become unhealthy, so the orchestrator maintains the desired state by creating another instance.
- Wraps the containers for each microservice in a simple service layer. The service layer:
  - Abstracts out complexities like IP address, port, and number of instances.
  - Load balances traffic between microservice instances.
  - Supports easy communication between dependent microservice instances.

Container orchestrators also provide flexibility and traffic control to:

- Release new versions or roll back to old versions of microservices or sets of microservices, without downtime.
- Enable side by side testing of different microservice versions.

Choose an Azure container orchestrator

Here are some options for implementing microservices container orchestration in Azure:

- Azure Kubernetes Service (AKS) is a fully managed Kubernetes container orchestration service in Azure that simplifies deployment and management of containerized applications. AKS provides elastic provisioning, fast end-to-end deployment, and advanced identity and access management.
- Azure Service Fabric is a container orchestrator for deploying and managing microservices across a cluster of machines. The lightweight Service Fabric runtime supports building stateless and stateful microservices.

  A key Service Fabric differentiator is its robust support for building stateful services. You can use the built-in stateful services programming model, or run containerized stateful services written in any language or code.
- Azure Container Instances (ACI) is the quickest and simplest way to run a container in Azure. With ACI, you don't have to manage virtual machines or adapt higher-level services.

  For simple orchestration scenarios, you can use Docker Compose to define and run a multi-container application locally. Then, deploy the Docker containers as an ACI container group in a managed, serverless Azure environment. For full container orchestration scenarios, ACI can integrate with AKS to create virtual nodes for AKS orchestration.
- Azure Spring Cloud is an enterprise-ready, fully managed service for Spring Boot apps. With Spring Cloud, you can focus on building and running apps without having to manage infrastructure. Spring Cloud comes with built-in lifecycle and orchestration management, ease of monitoring, and full integration with Azure.
- Azure Red Hat OpenShift (ARO) supports deployment of fully managed OpenShift clusters on Azure. Running Kubernetes production containers requires integration with frameworks and tools like image registries, storage management, monitoring, and DevOps. ARO extends Kubernetes by combining these components into a single container platform as a service (PaaS).

Next steps

- Microservices architecture on Azure Kubernetes Service (AKS)
- Advanced Azure Kubernetes Service (AKS) microservices architecture
- Microservices with AKS and Azure DevOps
- Use API gateways in microservices
- Monitor a microservices architecture in AKS
- Microservices architecture on Azure Service Fabric
- Azure Spring Cloud reference architecture

Related resources

- Build microservices on Azure
- Design a microservices architecture
- Design patterns for microservices
- Microservices architectural style
- Azure Kubernetes Service solution journey

Design patterns for microservices

11/19/2021 • 2 minutes to read • Edit Online

The goal of microservices is to increase the velocity of application releases, by decomposing the application into small autonomous services that can be deployed independently. A microservices architecture also brings some challenges. The design patterns shown here can help mitigate these challenges.

Ambassador can be used to offload common client connectivity tasks such as monitoring, logging, routing, and security (such as TLS) in a language agnostic way. Ambassador services are often deployed as a sidecar (see below).

Anti-corruption layer implements a façade between new and legacy applications, to ensure that the design of a new application is not limited by dependencies on legacy systems.

Backends for Frontends creates separate backend services for different types of clients, such as desktop and mobile. That way, a single backend service doesn't need to handle the conflicting requirements of various client types. This pattern can help keep each microservice simple, by separating client-specific concerns.

Bulkhead isolates critical resources, such as connection pool, memory, and CPU, for each workload or service. By using bulkheads, a single workload (or service) can't consume all of the resources, starving others. This pattern increases the resiliency of the system by preventing cascading failures caused by one service.

Gateway Aggregation aggregates requests to multiple individual microservices into a single request, reducing chattiness between consumers and services.

Gateway Offloading enables each microservice to offload shared service functionality, such as the use of SSL certificates, to an API gateway.

Gateway Routing routes requests to multiple microservices using a single endpoint, so that consumers don't need to manage many separate endpoints.

Sidecar deploys helper components of an application as a separate container or process to provide isolation and encapsulation.

Strangler Fig supports incremental refactoring of an application, by gradually replacing specific pieces of functionality with new services.

For the complete catalog of cloud design patterns on the Azure Architecture Center, see Cloud Design Patterns.

Monitoring a microservices architecture in Azure Kubernetes Service (AKS)

11/19/2021 • 14 minutes to read • Edit Online

This article describes best practices for monitoring a microservices application that runs on Azure Kubernetes Service (AKS).

In any complex application, at some point something will go wrong. In a microservices application, you need to track what's happening across dozens or even hundreds of services. To make sense of what's happening, you must collect telemetry from the application. Telemetry can be divided into *logs* and *metrics*.

Logs are text-based records of events that occur while the application is running. They include things like application logs (trace statements) or web server logs. Logs are primarily useful for forensics and root cause analysis.

Metrics are numerical values that can be analyzed. You can use them to observe the system in real time (or close to real time), or to analyze performance trends over time. To understand the system holistically, you must collect metrics at various levels of the architecture, from the physical infrastructure to the application, including:

- **Node-level** metrics, including CPU, memory, network, disk, and file system usage. System metrics help you to understand resource allocation for each node in the cluster, and troubleshoot outliers.
- **Container** metrics. For containerized applications, you need to collect metrics at the container level, not just at the VM level.
- **Application** metrics. This includes any metrics that are relevant to understanding the behavior of a service. Examples include the number of queued inbound HTTP requests, request latency, or message queue length. Applications can also create custom metrics that are specific to the domain, such as the number of business transactions processed per minute.
- **Dependent service** metrics. Services may call external services or endpoints, such as managed PaaS services or SaaS services. Third-party services may or may not provide any metrics. If not, you'll have to rely on your own application metrics to track statistics for latency and error rate.

Monitoring cluster status

Use Azure Monitor to monitor the overall health of your clusters. The following screenshot shows a cluster with critical errors in user-deployed pods.

From here, you can drill in further to find the issue. For example, if the pod status is `ImagePullBackoff`, it means that Kubernetes could not pull the container image from the registry. This could be caused by an invalid container tag or an authentication error trying to pull from the registry.

Note that a container crashing will put the container state into `State` = `Waiting`,with `Reason` = `CrashLoopBackOff`. For a typical scenario where a pod is part of a replica set and the retry policy is `Always`, this won't show as an error in the cluster status. However, you can run queries or set up alerts for this condition. For more information, see Understand AKS cluster performance with Azure Monitor for containers.

Metrics

We recommend using Azure Monitor to collect and view metrics for your AKS clusters and any other dependent Azure services.

- For cluster and container metrics, enable Azure Monitor for containers. When this feature is enabled, Azure Monitor collects memory and processor metrics from controllers, nodes, and containers via the Kubernetes metrics API. For more information about the metrics that are available through Azure Monitor for containers, see Understand AKS cluster performance with Azure Monitor for containers.

- Use Application Insights to collect application metrics. Application Insights is an extensible Application Performance Management (APM) service. To use it, you install an instrumentation package in your application. This package monitors the app and sends telemetry data to the Application Insights service. It can also pull telemetry data from the host environment. The data is then sent to Azure Monitor. Application Insights also provides built-in correlation and dependency tracking (see Distributed tracing, below).

Application Insights has a maximum throughput measured in events/second, and it throttles if the data rate exceeds the limit. For details, see Application Insights limits. Create different Application Insights instances per environment, so that dev/test environments don't compete against the production telemetry for quota.

A single operation may generate several telemetry events, so if the application experiences a high volume of traffic, it is likely to get throttled. To mitigate this problem, you can perform sampling to reduce the telemetry traffic. The tradeoff is that your metrics will be less precise. For more information, see Sampling in Application Insights. You can also reduce the data volume by pre-aggregating metrics — that is, calculating statistical values such as average and standard deviation, and sending those values instead of the raw telemetry. The following blog post describes an approach to using Application Insights at scale: Azure Monitoring and Analytics at Scale.

If your data rate is high enough to trigger throttling, and sampling or aggregation are not acceptable, consider exporting metrics to a time-series database such as **Prometheus** or **InfluxDB** running in the cluster.

- InfluxDB is a push-based system. An agent needs to push the metrics. You can use TICK stack, to setup monitoring of Kubernetes, and push it to InfluxDB using Telegraf, which is an agent for collecting and reporting metrics. InfluxDB can be used for irregular events and string data types.

- Prometheus is a pull-based system. It periodically scrapes metrics from configured locations. Prometheus can scrape metrics generated by cAdvisor or kube-state-metrics. kube-state-metrics is a service that collects metrics from the Kubernetes API server and makes them available to Prometheus (or a scraper that is compatible with a Prometheus client endpoint). For system metrics, use Node exporter, which is a Prometheus exporter for system metrics. Prometheus supports floating point data, but not string data, so it is appropriate for system metrics but not logs. Kubernetes Metrics Server is a cluster-wide aggregator of resource usage data.

Logging

Here are some of the general challenges of logging in a microservices application:

- Understanding the end-to-end processing of a client request, where multiple services might be invoked to handle a single request.
- Consolidating logs from multiple services into a single aggregated view.

- Parsing logs that come from multiple sources, which use their own logging schemas or have no particular schema. Logs may be generated by third-party components that you don't control.
- Microservices architectures often generate a larger volume of logs than traditional monoliths, because there are more services, network calls, and steps in a transaction. That means logging itself can be a performance or resource bottleneck for the application.

There are some additional challenges for a Kubernetes-based architecture:

- Containers can move around and be rescheduled.
- Kubernetes has a networking abstraction that uses virtual IP addresses and port mappings.

In Kubernetes, the standard approach to logging is for a container to write logs to stdout and stderr. The container engine redirects these streams to a logging driver. For ease of querying, and to prevent possible loss of log data if a node crashes, the usual approach is to collect the logs from each node and send them to a central storage location.

Azure Monitor integrates with AKS to support this approach. Azure Monitor collects container logs and sends them to a Log Analytics workspace. From there, you can use the Kusto query language to write queries across the aggregated logs. For example, here is a Kusto query to show the container logs for a specified pod:

```
let ContainerIdList = KubePodInventory
| where ClusterName =~ '<cluster-name>'
| where Name =~ '<pod-name>'
| distinct ContainerID;
ContainerLog
| where ContainerID in (ContainerIdList)
```

Azure Monitor is a managed service, and configuring an AKS cluster to use Azure Monitor is a simple configuration switch in the CLI or Resource Manager template. (For more information, see How to enable Azure Monitor for containers.) Another advantage of using Azure Monitoring is that it consolidates your AKS logs with other Azure platform logs, providing a unified monitoring experience.

Azure Monitor is billed per gigabyte (GB) of data ingested into the service (see Azure Monitor pricing). At very high volumes, cost may become a consideration. There are many open-source alternatives available for the Kubernetes ecosystem. For example, many organizations use **Fluentd** with **Elasticsearch**. Fluentd is an open-source data collector, and Elasticsearch is a document database that is for search. A challenge with these options is that they require additional configuration and management of the cluster. For a production workload, you may need to experiment with configuration settings. You'll also need to monitor the performance of the logging infrastructure.

Application Insights

For richer log data, we recommend instrumenting your code with Application Insights. This requires adding an Application Insights package to your code and configuring your code to send logging statements to Application Insights. The details depend on the platform, such as .NET, Java, or Node.js. The Application Insights package sends telemetry data to Azure Monitor.

If you are using .NET Core, we recommend also using the Application Insights for Kubernetes library. This library enriches Application Insights traces with additional information such as the container, node, pod, labels, and replica set.

Advantages of this approach include:

- Application Insights logs HTTP requests, including latency and result code.
- Distributed tracing is enabled by default.
- Traces include an operation ID, so you can match all traces for a particular operation.
- Traces generated by Application Insights often have additional contextual information. For example, ASP.NET

traces are decorated with the action name and a category such as `ControllerActionInvoker`, which give you insights into the ASP.NET request pipeline.

- Application Insights collects performance metrics for performance troubleshooting and optimization.

Considerations:

- Application Insights throttles the telemetry if the data rate exceeds a maximum limit; for details, see Application Insights limits. A single operation may generate several telemetry events, so if the application experiences a high volume of traffic, it is likely to get throttled.
- Because Application Insights batches data, it's possible to lose a batch if a process crashes with an unhandled exception.
- Application Insights is billed based on data volume. For more information, see Manage pricing and data volume in Application Insights.

Structured logging

To make logs easier to parse, use structured logging where possible. Structured logging is approach where the application writes logs in a structured format, such as JSON, rather than outputting unstructured text strings. There are many structured logging libraries available. For example, here is a logging statement that uses the Serilog library for .NET Core:

```
public async Task<IActionResult> Put([FromBody]Delivery delivery, string id)
{
    logger.LogInformation("In Put action with delivery {Id}: {@DeliveryInfo}", id, delivery.ToLogInfo());

    ...
}
```

Here, the call to `LogInformation` includes an `Id` parameter and `DeliveryInfo` parameter. With structured logging, these values are not interpolated into the message string. Instead, the log output will look something like this:

```
{"@t":"2019-06-13T00:57:09.9932697Z","@mt":"In Put action with delivery {Id}:
{@DeliveryInfo}","Id":"36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef","DeliveryInfo":{...
```

This is a JSON string, where the "@t" field is a timestamp, "@mt" is the message string, and the remaining key/value pairs are the parameters. Outputting JSON format makes it easier to query the data in a structured way. For example, the following Log Analytics query, written in the Kusto query language, searches for instances of this particular message from all containers named `fabrikam-delivery`:

```
traces
| where customDimensions.["Kubernetes.Container.Name"] == "fabrikam-delivery"
| where customDimensions.["{OriginalFormat}"] == "In Put action with delivery {Id}: {@DeliveryInfo}"
| project message, customDimensions["Id"], customDimensions["@DeliveryInfo"]
```

Viewing the result in the Azure portal shows that `DeliveryInfo` is a structured record that contains the serialized representation of the `DeliveryInfo` model:

message	customDimensions_Id	customDimensions_@DeliveryInfo
In Put action with delivery 36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef: {"Id":"36585f...	36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef	{"Id":"36585f2d-c1fa-4a3d-9e06-a7f4

message	In Put action with delivery 36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef: {"Id":"36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef","Owne
customDimensions_Id	36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef
customDimensions_@DeliveryInfo	{"Id":"36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef","Owner":{"UserId":"user id for logging","AccountId":"52dadf0c-0067-43e7-
ConfirmationRequired	0
Deadline	string
DroneId	AssignedDroneId01ba4d0b-c01a-4369-ba75-51bde0e76cc9
Dropoff	{"Altitude":0.31507750848078986,"Latitude":0.753494655598651,"Longitude":0.8935283077384942}
Expedited	true
Id	36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef
Owner	{"UserId":"user id for logging","AccountId":"52dadf0c-0067-43e7-af76-86e32b48bc5e"}
Pickup	{"Altitude":0.2929516161293497,"Latitude":0.26815900219052985,"Longitude":0.7984184430904773}

Here is the JSON from this example:

```
{
    "Id": "36585f2d-c1fa-4a3d-9e06-a7f40b7d04ef",
    "Owner": {
        "UserId": "user id for logging",
        "AccountId": "52dadf0c-0067-43e7-af76-86e32b48bc5e"
    },
    "Pickup": {
        "Altitude": 0.29295161612934972,
        "Latitude": 0.26815900219052985,
        "Longitude": 0.79841844309047727
    },
    "Dropoff": {
        "Altitude": 0.31507750848078986,
        "Latitude": 0.753494655598651,
        "Longitude": 0.89352830773849423
    },
    "Deadline": "string",
    "Expedited": true,
    "ConfirmationRequired": 0,
    "DroneId": "AssignedDroneId01ba4d0b-c01a-4369-ba75-51bde0e76cc9"
}
```

The previous code snippet used the Serilog library, but structured logging libraries are available for other languages as well. For example, here's an example using the SLF4J library for Java:

```
MDC.put("DeliveryId", deliveryId);

log.info("In schedule delivery action with delivery request {}", externalDelivery.toString());
```

Distributed tracing

A significant challenge of microservices is to understand the flow of events across services. A single transaction may involve calls to multiple services. To reconstruct the entire sequence of steps, each service should propagate a *correlation ID* that acts as a unique identifier for that operation. The correlation ID enables distributed tracing across services.

The first service that receives a client request should generate the correlation ID. If the service makes an HTTP call to another service, it puts the correlation ID in a request header. If the service sends an asynchronous message, it puts the correlation ID into the message. Downstream services continue to propagate the correlation ID, so that it flows through the entire system. In addition, all code that writes application metrics or log events should include the correlation ID.

When service calls are correlated, you can calculate operational metrics such as the end-to-end latency for a complete transaction, the number of successful transactions per second, and the percentage of failed transactions. Including correlation IDs in application logs makes it possible to perform root cause analysis. If an operation fails, you can find the log statements for all of the service calls that were part of the same operation.

We recommend using Application Insights for distributed tracing. The Application Insights SDK automatically injects correlation context into HTTP headers, and includes the correlation ID in Application Insights logs. Some services may still need to explicitly propagate the correlation headers, depending on the frameworks and libraries being used. For more information, see Telemetry correlation in Application Insights.

Some additional considerations when implementing distributed tracing:

- There is now a standard HTTP header for correlation IDs, a W3C proposal has been accepted as an official recommendation recently. Your team should standardize on a custom header value. The choice may be decided by your logging framework, such as Application Insights, or choice of service mesh.
- For asynchronous messages, if your messaging infrastructure supports adding metadata to messages, you should include the correlation ID as metadata. Otherwise, include it as part of the message schema. For example, see Distributed tracing and correlation through Service Bus messaging.
- Rather than a single opaque identifier, you might send a *correlation context* that includes richer information, such as caller-callee relationships.
- If you are using Istio or linkerd as a service mesh, these technologies automatically generate correlation headers when HTTP calls are routed through the service mesh proxies. Services should forward the relevant headers.

Example of distributed tracing

This example follows a distributed transaction through a set of microservices. The example is taken from a reference implementation described here.

In this scenario, the distributed transaction has the following steps:

1. The Ingestion service puts a message on a Service Bus queue.
2. The Workflow service pulls the message from the queue.
3. The Workflow service calls three backend services to process the request (Drone Scheduler, Package, and Delivery).

The following screenshot shows the application map for the Drone Delivery application. This map shows calls to the public API endpoint that result in a workflow that involves five microservices.

The arrows from `fabrikam-workflow` and `fabrikam-ingestion` to a Service Bus queue show where the messages are sent and received. You can't tell from the diagram which service is sending messages and which is receiving — the arrows just show that both services are calling Service Bus — but this information is available in the details:

Because every call includes an operation ID, you can also view the end-to-end steps in a single transaction, including timing information and the HTTP calls at each step. Here is the visualization of one such transaction:

This visualization shows the steps from the ingestion service to the queue, from the queue to the workflow service, and from the workflow service to the other backend services. The last step is the Workflow service marking the Service Bus message as completed.

Now here is an example when calls to a backend service were failing:

This shows that a large fraction (36%) of calls to the Drone Scheduler service failed during the period being queried. In the end-to-end transaction view, it shows that an exception occurs when sending an HTTP PUT request to the service.

Further drilling in, the exception turns out to be a socket exception, "No such device or address."

```
Fabrikam.Workflow.Service.Services.BackendServiceCallFailedException: No such device or address ---u003e
System.Net.Http.HttpRequestException: No such device or address ---u003e
System.Net.Sockets.SocketException: No such device or address
```

This is a hint that the backend service is not reachable. At this point, you might use kubectl to view the deployment configuration. In this example, it turned out the service hostname was not resolving, due to an error in the Kubernetes configuration files. The article Debug Services in the Kubernetes documentation has tips for diagnosing this sort of error.

Here are some common causes of errors:

- Code bugs. These might manifest as:
  - Exceptions. Look in the Application Insights logs to view the exception details.
  - Process crashing. Look at container and pod status, and view container logs or Application Insights traces.
  - HTTP 5xx errors
- Resource exhaustion:
  - Look for throttling (HTTP 429) or request timeouts.
  - Examine container metrics for CPU, memory, and disk
  - Look at the configurations for container and pod resource limits.
- Service discovery. Examine the Kubernetes service configuration and port mappings.
- API mismatch. Look for HTTP 400 errors. If APIs are versioned, look at which version is being called.
- Error pulling a container image. Look at the pod specification. Also make sure the cluster is authorized to pull from the container registry.
- RBAC issues.

Next steps

Learn more about features in Azure Monitor that support monitoring of applications on AKS:

- Azure Monitor for containers overview

- Understand AKS cluster performance with Azure Monitor for containers

For more information about using metrics for performance tuning, see see Performance tuning a distributed application.

CI/CD for microservices architectures

11/19/2021 • 8 minutes to read • Edit Online

Faster release cycles are one of the major advantages of microservices architectures. But without a good CI/CD process, you won't achieve the agility that microservices promise. This article describes the challenges and recommends some approaches to the problem.

What is CI/CD?

When we talk about CI/CD, we're really talking about several related processes: Continuous integration, continuous delivery, and continuous deployment.

- **Continuous integration**. Code changes are frequently merged into the main branch. Automated build and test processes ensure that code in the main branch is always production-quality.
- **Continuous delivery**. Any code changes that pass the CI process are automatically published to a production-like environment. Deployment into the live production environment may require manual approval, but is otherwise automated. The goal is that your code should always be *ready* to deploy into production.
- **Continuous deployment**. Code changes that pass the previous two steps are automatically deployed *into production*.

Here are some goals of a robust CI/CD process for a microservices architecture:

- Each team can build and deploy the services that it owns independently, without affecting or disrupting other teams.
- Before a new version of a service is deployed to production, it gets deployed to dev/test/QA environments for validation. Quality gates are enforced at each stage.
- A new version of a service can be deployed side by side with the previous version.
- Sufficient access control policies are in place.
- For containerized workloads, you can trust the container images that are deployed to production.

Why a robust CI/CD pipeline matters

In a traditional monolithic application, there is a single build pipeline whose output is the application executable. All development work feeds into this pipeline. If a high-priority bug is found, a fix must be integrated, tested, and published, which can delay the release of new features. You can mitigate these problems by having well-factored modules and using feature branches to minimize the impact of code changes. But as the application grows more complex, and more features are added, the release process for a monolith tends to become more brittle and likely to break.

Following the microservices philosophy, there should never be a long release train where every team has to get in line. The team that builds service "A" can release an update at any time, without waiting for changes in service "B" to be merged, tested, and deployed.

To achieve a high release velocity, your release pipeline must be automated and highly reliable to minimize risk. If you release to production one or more times daily , regressions or service disruptions must be rare. At the same time, if a bad update does get deployed, you must have a reliable way to quickly roll back or roll forward to a previous version of a service.

Challenges

- **Many small independent code bases**. Each team is responsible for building its own service, with its own build pipeline. In some organizations, teams may use separate code repositories. Separate repositories can lead to a situation where the knowledge of how to build the system is spread across teams, and nobody in the organization knows how to deploy the entire application. For example, what happens in a disaster recovery scenario, if you need to quickly deploy to a new cluster?

  Mitigation: Have a unified and automated pipeline to build and deploy services, so that this knowledge is not "hidden" within each team.

- **Multiple languages and frameworks**. With each team using its own mix of technologies, it can be difficult to create a single build process that works across the organization. The build process must be flexible enough that every team can adapt it for their choice of language or framework.

  Mitigation: Containerize the build process for each service. That way, the build system just needs to be able to run the containers.

- **Integration and load testing**. With teams releasing updates at their own pace, it can be challenging to design robust end-to-end testing, especially when services have dependencies on other services. Moreover, running a full production cluster can be expensive, so it's unlikely that every team will run its own full cluster at production scales, just for testing.

- **Release management**. Every team should be able to deploy an update to production. That doesn't mean that every team member has permissions to do so. But having a centralized Release Manager role can reduce the velocity of deployments.

  Mitigation: The more that your CI/CD process is automated and reliable, the less there should be a need for a central authority. That said, you might have different policies for releasing major feature updates versus minor bug fixes. Being decentralized doesn't mean zero governance.

- **Service updates**. When you update a service to a new version, it shouldn't break other services that depend on it.

  Mitigation: Use deployment techniques such as blue-green or canary release for non-breaking changes. For breaking API changes, deploy the new version side by side with the previous version. That way, services that consume the previous API can be updated and tested for the new API. See Updating services, below.

Monorepo vs. multi-repo

Before creating a CI/CD workflow, you must know how the code base will be structured and managed.

- Do teams work in separate repositories or in a monorepo (single repository)?

- What is your branching strategy?
- Who can push code to production? Is there a release manager role?

The monorepo approach has been gaining favor but there are advantages and disadvantages to both.

	MONOREPO	MULTIPLE REPOS
Advantages	Code sharing Easier to standardize code and tooling Easier to refactor code Discoverability - single view of the code	Clear ownership per team Potentially fewer merge conflicts Helps to enforce decoupling of microservices
Challenges	Changes to shared code can affect multiple microservices Greater potential for merge conflicts Tooling must scale to a large code base Access control More complex deployment process	Harder to share code Harder to enforce coding standards Dependency management Diffuse code base, poor discoverability Lack of shared infrastructure

Updating services

There are various strategies for updating a service that's already in production. Here we discuss three common options: Rolling update, blue-green deployment, and canary release.

Rolling updates

In a rolling update, you deploy new instances of a service, and the new instances start receiving requests right away. As the new instances come up, the previous instances are removed.

Example. In Kubernetes, rolling updates are the default behavior when you update the pod spec for a Deployment. The Deployment controller creates a new ReplicaSet for the updated pods. Then it scales up the new ReplicaSet while scaling down the old one, to maintain the desired replica count. It doesn't delete old pods until the new ones are ready. Kubernetes keeps a history of the update, so you can roll back an update if needed.

Example. Azure Service Fabric uses the rolling update strategy by default. This strategy is best suited for deploying a version of a service with new features without changing existing APIs. Service Fabric starts an upgrade deployment by updating the application type to a subset of the nodes or an update domain. It then rolls forward to the next update domain until all domains are upgraded. If an upgrade domain fails to update, the application type rolls back to the previous version across all domains. Be aware that an application type with multiple services (and if all services are updated as part of one upgrade deployment) is prone to failure. If one service fails to update, the entire application is rolled back to the previous version and the other services are not updated.

One challenge of rolling updates is that during the update process, a mix of old and new versions are running and receiving traffic. During this period, any request could get routed to either of the two versions.

For breaking API changes, a good practice is to support both versions side by side, until all clients of the previous version are updated. See API versioning.

Blue-green deployment

In a blue-green deployment, you deploy the new version alongside the previous version. After you validate the new version, you switch all traffic at once from the previous version to the new version. After the switch, you monitor the application for any problems. If something goes wrong, you can swap back to the old version. Assuming there are no problems, you can delete the old version.

With a more traditional monolithic or N-tier application, blue-green deployment generally meant provisioning two identical environments. You would deploy the new version to a staging environment, then redirect client

traffic to the staging environment — for example, by swapping VIP addresses. In a microservices architecture, updates happen at the microservice level, so you would typically deploy the update into the same environment and use a service discovery mechanism to swap.

Example. In Kubernetes, you don't need to provision a separate cluster to do blue-green deployments. Instead, you can take advantage of selectors. Create a new Deployment resource with a new pod spec and a different set of labels. Create this deployment, without deleting the previous deployment or modifying the service that points to it. Once the new pods are running, you can update the service's selector to match the new deployment.

One drawback of blue-green deployment is that during the update, you are running twice as many pods for the service (current and next). If the pods require a lot of CPU or memory resources, you may need to scale out the cluster temporarily to handle the resource consumption.

Canary release

In a canary release, you roll out an updated version to a small number of clients. Then you monitor the behavior of the new service before rolling it out to all clients. This lets you do a slow rollout in a controlled fashion, observe real data, and spot problems before all customers are affected.

A canary release is more complex to manage than either blue-green or rolling update, because you must dynamically route requests to different versions of the service.

Example. In Kubernetes, you can configure a Service to span two replica sets (one for each version) and adjust the replica counts manually. However, this approach is rather coarse-grained, because of the way Kubernetes load balances across pods. For example, if you have a total of 10 replicas, you can only shift traffic in 10% increments. If you are using a service mesh, you can use the service mesh routing rules to implement a more sophisticated canary release strategy.

Next steps

Learn specific CI/CD practices for microservices running on Kubernetes.

- CI/CD for microservices on Kubernetes

Building a CI/CD pipeline for microservices on Kubernetes

11/19/2021 • 12 minutes to read • Edit Online

It can be challenging to create a reliable CI/CD process for a microservices architecture. Individual teams must be able to release services quickly and reliably, without disrupting other teams or destabilizing the application as a whole.

This article describes an example CI/CD pipeline for deploying microservices to Azure Kubernetes Service (AKS). Every team and project is different, so don't take this article as a set of hard-and-fast rules. Instead, it's meant to be a starting point for designing your own CI/CD process.

The goals of a CI/CD pipeline for Kubernetes hosted microservices can be summarized as follows:

- Teams can build and deploy their services independently.
- Code changes that pass the CI process are automatically deployed to a production-like environment.
- Quality gates are enforced at each stage of the pipeline.
- A new version of a service can be deployed side by side with the previous version.

For more background, see CI/CD for microservices architectures.

Assumptions

For purposes of this example, here are some assumptions about the development team and the code base:

- The code repository is a monorepo, with folders organized by microservice.
- The team's branching strategy is based on trunk-based development.
- The team uses release branches to manage releases. Separate releases are created for each microservice.
- The CI/CD process uses Azure Pipelines to build, test, and deploy the microservices to AKS.
- The container images for each microservice are stored in Azure Container Registry.
- The team uses Helm charts to package each microservice.

These assumptions drive many of the specific details of the CI/CD pipeline. However, the basic approach described here be adapted for other processes, tools, and services, such as Jenkins or Docker Hub.

Validation builds

Suppose that a developer is working on a microservice called the Delivery Service. While developing a new feature, the developer checks code into a feature branch. By convention, feature branches are named `feature/*`.

The build definition file includes a trigger that filters by the branch name and the source path:

```
trigger:
  batch: true
  branches:
    include:
    # for new release to production: release flow strategy
    - release/delivery/v*
    - refs/release/delivery/v*
    - master
    - feature/delivery/*
    - topic/delivery/*
  paths:
    include:
    - /src/shipping/delivery/
```

Using this approach, each team can have its own build pipeline. Only code that is checked into the `/src/shipping/delivery` folder triggers a build of the Delivery Service. Pushing commits to a branch that matches the filter triggers a CI build. At this point in the workflow, the CI build runs some minimal code verification:

1. Build the code.
2. Run unit tests.

The goal is to keep build times short so that the developer can get quick feedback. Once the feature is ready to merge into master, the developer opens a PR. This operation triggers another CI build that performs some additional checks:

1. Build the code.
2. Run unit tests.
3. Build the runtime container image.
4. Run vulnerability scans on the image.

> **NOTE**
> In Azure DevOps Repos, you can define policies to protect branches. For example, the policy could require a successful CI build plus a sign-off from an approver in order to merge into master.

Full CI/CD build

At some point, the team is ready to deploy a new version of the Delivery service. The release manager creates a branch from the main branch with this naming pattern: `release/<microservice name>/<semver>`. For example, `release/delivery/v1.0.2`.

The creation of this branch triggers a full CI build that runs all of the previous steps plus:

1. Push the container image to Azure Container Registry. The image is tagged with the version number taken from the branch name.
2. Run `helm package` to package the Helm chart for the service. The chart is also tagged with a version number.
3. Push the Helm package to Container Registry.

Assuming this build succeeds, it triggers a deployment (CD) process using an Azure Pipelines release pipeline. This pipeline has the following steps:

1. Deploy the Helm chart to a QA environment.
2. An approver signs off before the package moves to production. See Release deployment control using approvals.
3. Retag the Docker image for the production namespace in Azure Container Registry. For example, if the current tag is `myrepo.azurecr.io/delivery:v1.0.2`, the production tag is `myrepo.azurecr.io/prod/delivery:v1.0.2`.
4. Deploy the Helm chart to the production environment.

Even in a monorepo, these tasks can be scoped to individual microservices so that teams can deploy with high velocity. The process has some manual steps: Approving PRs, creating release branches, and approving deployments into the production cluster. These steps are manual; they could be automated if the organization prefers.

Isolation of environments

You will have multiple environments where you deploy services, including environments for development, smoke testing, integration testing, load testing, and finally, production. These environments need some level of isolation. In Kubernetes, you have a choice between physical isolation and logical isolation. Physical isolation means deploying to separate clusters. Logical isolation uses namespaces and policies, as described earlier.

Our recommendation is to create a dedicated production cluster along with a separate cluster for your dev/test environments. Use logical isolation to separate environments within the dev/test cluster. Services deployed to the dev/test cluster should never have access to data stores that hold business data.

Build process

When possible, package your build process into a Docker container. This configuration allows you to build code artifacts using Docker and without configuring a build environment on each build machine. A containerized build process makes it easy to scale out the CI pipeline by adding new build agents. Also, any developer on the team can build the code simply by running the build container.

By using multi-stage builds in Docker, you can define the build environment and the runtime image in a single Dockerfile. For example, here's a Dockerfile that builds a .NET application:

```
FROM mcr.microsoft.com/dotnet/core/runtime:3.1 AS base
WORKDIR /app

FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src/Fabrikam.Workflow.Service

COPY Fabrikam.Workflow.Service/Fabrikam.Workflow.Service.csproj .
RUN dotnet restore Fabrikam.Workflow.Service.csproj

COPY Fabrikam.Workflow.Service/. .
RUN dotnet build Fabrikam.Workflow.Service.csproj -c release -o /app --no-restore

FROM build AS testrunner
WORKDIR /src/tests

COPY Fabrikam.Workflow.Service.Tests/*.csproj .
RUN dotnet restore Fabrikam.Workflow.Service.Tests.csproj

COPY Fabrikam.Workflow.Service.Tests/. .
ENTRYPOINT ["dotnet", "test", "--logger:trx"]

FROM build AS publish
RUN dotnet publish Fabrikam.Workflow.Service.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Fabrikam.Workflow.Service.dll"]
```

This Dockerfile defines several build stages. Notice that the stage named `base` uses the .NET runtime, while the stage named `build` uses the full .NET SDK. The `build` stage is used to build the .NET project. But the final runtime container is built from `base`, which contains just the runtime and is significantly smaller than the full SDK image.

Building a test runner

Another good practice is to run unit tests in the container. For example, here is part of a Docker file that builds a test runner:

```
FROM build AS testrunner
WORKDIR /src/tests

COPY Fabrikam.Workflow.Service.Tests/*.csproj .
RUN dotnet restore Fabrikam.Workflow.Service.Tests.csproj

COPY Fabrikam.Workflow.Service.Tests/. .
ENTRYPOINT ["dotnet", "test", "--logger:trx"]
```

A developer can use this Docker file to run the tests locally:

```
docker build . -t delivery-test:1 --target=testrunner
docker run delivery-test:1
```

The CI pipeline should also run the tests as part of the build verification step.

Note that this file uses the Docker `ENTRYPOINT` command to run the tests, not the Docker `RUN` command.

- If you use the `RUN` command, the tests run every time you build the image. By using `ENTRYPOINT`, the tests are opt-in. They run only when you explicitly target the `testrunner` stage.
- A failing test doesn't cause the Docker `build` command to fail. That way, you can distinguish container build

failures from test failures.
- Test results can be saved to a mounted volume.

Container best practices

Here are some other best practices to consider for containers:

- Define organization-wide conventions for container tags, versioning, and naming conventions for resources deployed to the cluster (pods, services, and so on). That can make it easier to diagnose deployment issues.
- During the development and test cycle, the CI/CD process will build many container images. Only some of those images are candidates for release, and then only some of those release candidates will get promoted to production. Have a clear versioning strategy so that you know which images are currently deployed to production and to help roll back to a previous version if necessary.
- Always deploy specific container version tags, not `latest`.
- Use namespaces in Azure Container Registry to isolate images that are approved for production from images that are still being tested. Don't move an image into the production namespace until you're ready to deploy it into production. If you combine this practice with semantic versioning of container images, it can reduce the chance of accidentally deploying a version that wasn't approved for release.
- Follow the principle of least privilege by running containers as a nonprivileged user. In Kubernetes, you can create a pod security policy that prevents containers from running as *root*. See Prevent Pods From Running With Root Privileges.

Helm charts

Consider using Helm to manage building and deploying services. Here are some of the features of Helm that help with CI/CD:

- Often, a single microservice is defined by multiple Kubernetes objects. Helm allows these objects to be packaged into a single Helm chart.
- A chart can be deployed with a single Helm command rather than a series of kubectl commands.
- Charts are explicitly versioned. Use Helm to release a version, view releases, and roll back to a previous version. Tracking updates and revisions, using semantic versioning, along with the ability to roll back to a previous version.
- Helm charts use templates to avoid duplicating information, such as labels and selectors, across many files.
- Helm can manage dependencies between charts.
- Charts can be stored in a Helm repository, such as Azure Container Registry, and integrated into the build pipeline.

For more information about using Container Registry as a Helm repository, see Use Azure Container Registry as a Helm repository for your application charts.

> **IMPORTANT**
>
> This feature is currently in preview. Previews are made available to you on the condition that you agree to the supplemental terms of use. Some aspects of this feature may change prior to general availability (GA).

A single microservice may involve multiple Kubernetes configuration files. Updating a service can mean touching all of these files to update selectors, labels, and image tags. Helm treats these as a single package called a chart and allows you to easily update the YAML files by using variables. Helm uses a template language (based on Go templates) to let you write parameterized YAML configuration files.

For example, here's part of a YAML file that defines a deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "package.fullname" . | replace "." "" }}
  labels:
    app.kubernetes.io/name: {{ include "package.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
  annotations:
    kubernetes.io/change-cause: {{ .Values.reason }}

...

  spec:
      containers:
      - name: &package-container_name fabrikam-package
        image: {{ .Values.dockerregistry }}/{{ .Values.image.repository }}:{{ .Values.image.tag }}
        imagePullPolicy: {{ .Values.image.pullPolicy }}
        env:
        - name: LOG_LEVEL
          value: {{ .Values.log.level }}
```

You can see that the deployment name, labels, and container spec all use template parameters, which are provided at deployment time. For example, from the command line:

```
helm install $HELM_CHARTS/package/ \
     --set image.tag=0.1.0 \
     --set image.repository=package \
     --set dockerregistry=$ACR_SERVER \
     --namespace backend \
     --name package-v0.1.0
```

Although your CI/CD pipeline could install a chart directly to Kubernetes, we recommend creating a chart archive (.tgz file) and pushing the chart to a Helm repository such as Azure Container Registry. For more information, see Package Docker-based apps in Helm charts in Azure Pipelines.

Consider deploying Helm to its own namespace and using role-based access control (RBAC) to restrict which namespaces it can deploy to. For more information, see Role-based Access Control in the Helm documentation.

Revisions

Helm charts always have a version number, which must use semantic versioning. A chart can also have an `appVersion`. This field is optional and doesn't have to be related to the chart version. Some teams might want to application versions separately from updates to the charts. But a simpler approach is to use one version number, so there's a 1:1 relation between chart version and application version. That way, you can store one chart per release and easily deploy the desired release:

```
helm install <package-chart-name> --version <desiredVersion>
```

Another good practice is to provide a change-cause annotation in the deployment template:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "delivery.fullname" . | replace "." "" }}
  labels:
     ...
  annotations:
    kubernetes.io/change-cause: {{ .Values.reason }}
```

This lets you view the change-cause field for each revision, using the `kubectl rollout history` command. In the previous example, the change-cause is provided as a Helm chart parameter.

```
kubectl rollout history deployments/delivery-v010 -n backend
```

```
deployment.extensions/delivery-v010
REVISION  CHANGE-CAUSE
1         Initial deployment
```

You can also use the `helm list` command to view the revision history:

```
helm list
```

```
NAME            REVISION    UPDATED                     STATUS        CHART            APP VERSION
NAMESPACE
delivery-v0.1.0 1           Sun Apr  7 00:25:30 2020    DEPLOYED      delivery-v0.1.0  v0.1.0
backend
```

Azure DevOps Pipeline

In Azure Pipelines, pipelines are divided into *build pipelines* and *release pipelines*. The build pipeline runs the CI process and creates build artifacts. For a microservices architecture on Kubernetes, these artifacts are the container images and Helm charts that define each microservice. The release pipeline runs that CD process that deploys a microservice into a cluster.

Based on the CI flow described earlier in this article, a build pipeline might consist of the following tasks:

1. Build the test runner container.

   ```
   - task: Docker@1
     inputs:
       azureSubscriptionEndpoint: $(AzureSubscription)
       azureContainerRegistry: $(AzureContainerRegistry)
       arguments: '--pull --target testrunner'
       dockerFile: $(System.DefaultWorkingDirectory)/$(dockerFileName)
       imageName: '$(imageName)-test'
   ```

2. Run the tests, by invoking docker run against the test runner container.

```
- task: Docker@1
  inputs:
    azureSubscriptionEndpoint: $(AzureSubscription)
    azureContainerRegistry: $(AzureContainerRegistry)
    command: 'run'
    containerName: testrunner
    volumes: '$(System.DefaultWorkingDirectory)/TestResults:/app/tests/TestResults'
    imageName: '$(imageName)-test'
    runInBackground: false
```

3. Publish the test results. See Build an image.

```
- task: PublishTestResults@2
  inputs:
    testResultsFormat: 'VSTest'
    testResultsFiles: 'TestResults/*.trx'
    searchFolder: '$(System.DefaultWorkingDirectory)'
    publishRunAttachments: true
```

4. Build the runtime container.

```
- task: Docker@1
  inputs:
    azureSubscriptionEndpoint: $(AzureSubscription)
    azureContainerRegistry: $(AzureContainerRegistry)
    dockerFile: $(System.DefaultWorkingDirectory)/$(dockerFileName)
    includeLatestTag: false
    imageName: '$(imageName)'
```

5. Push the container image to Azure Container Registry (or other container registry).

```
- task: Docker@1
  inputs:
    azureSubscriptionEndpoint: $(AzureSubscription)
    azureContainerRegistry: $(AzureContainerRegistry)
    command: 'Push an image'
    imageName: '$(imageName)'
    includeSourceTags: false
```

6. Package the Helm chart.

```
- task: HelmDeploy@0
  inputs:
    command: package
    chartPath: $(chartPath)
    chartVersion: $(Build.SourceBranchName)
    arguments: '--app-version $(Build.SourceBranchName)'
```

7. Push the Helm package to Azure Container Registry (or other Helm repository).

```
task: AzureCLI@1
  inputs:
    azureSubscription: $(AzureSubscription)
    scriptLocation: inlineScript
    inlineScript: |
    az acr helm push $(System.ArtifactsDirectory)/$(repositoryName)-$(Build.SourceBranchName).tgz --
name $(AzureContainerRegistry);
```

The output from the CI pipeline is a production-ready container image and an updated Helm chart for the microservice. At this point, the release pipeline can take over. It performs the following steps:

- Deploy to dev/QA/staging environments.
- Wait for an approver to approve or reject the deployment.
- Retag the container image for release
- Push the release tag to the container registry.
- Upgrade the Helm chart in the production cluster.

For more information about creating a release pipeline, see Release pipelines, draft releases, and release options.

The following diagram shows the end-to-end CI/CD process described in this article:

Migrate a monolith application to microservices using domain-driven design

11/19/2021 • 6 minutes to read • Edit Online

This article describes how to use domain-driven design (DDD) to migrate a monolithic application to microservices.

A monolithic application is typically an application system in which all of the relevant modules are packaged together as a single deployable unit of execution. For example, it might be a Java Web Application (WAR) running on Tomcat or an ASP.NET application running on IIS. A typical monolithic application uses a layered design, with separate layers for UI, application logic, and data access.

These systems start small but tend to grow over time to meet business needs. At some point, as new features are added, a monolithic application can begin to suffer from the following problems:

- The individual parts of the system cannot be scaled independently, because they are tightly coupled.
- It is hard to maintain the code, because of tight coupling and hidden dependencies.
- Testing becomes harder, increasing the probability of introducing vulnerabilities.

These problems can become an obstacle to future growth and stability. Teams become wary of making changes, especially if the original developers are no longer working on the project and design documents are sparse or outdated.

Despite these limitations, a monolithic design can make sense as a starting point for an application. Monoliths are often the quickest path to building a proof-of-concept or minimal viable product. In the early phases of development, monoliths tend to be:

- Easier to build, because there is a single shared code base.
- Easier to debug, because the code runs within a single process and memory space.
- Easier to reason about, because there are fewer moving parts.

As the application grows in complexity, however, these advantages can disappear. Large monoliths often become progressively harder to build, debug, and reason about. At some point, the problems outweigh the benefits. This is the point when it can make sense to migrate the application to a microservices architecture. Unlike monoliths, microservices are typically decentralized, loosely coupled units of execution. The following diagram shows a

typical microservices architecture:

Migrating a monolith to a microservice requires significant time and investment to avoid failures or overruns. To ensure that any migration is successful, it's good to understand both the benefits and also challenges that microservices bring. The benefits include:

- Services can evolve independently based on user needs.
- Services can scale independently to meet user demand.
- Over time, development cycles become faster as features can be released to market quicker.
- Services are isolated and are more tolerant of failure.
- A single service that fails will not bring down the entire application.
- Testing becomes more coherent and consistent, using behavior-driven development.

For more information about the benefits and challenges of microservices, see Microservices architecture style.

Apply domain-driven design

Any migration strategy should allow teams to incrementally refactor the application into smaller services, while still providing continuity of service to end users. Here's the general approach:

- Stop adding functionality to the monolith.
- Split the front end from the back end.
- Decompose and decouple the monolith into a series of microservices.

To help facilitate this decomposition, a viable software development approach is to apply the principles of domain-driven design (DDD).

Domain Driven Design (DDD) is a software development approach first introduced by Eric Evans. DDD requires a good understanding of the domain for which the application will be written. The necessary domain knowledge to create the application resides within the people who understand it — the domain experts.

The DDD approach can be applied retroactively to an existing application, as a way to begin decomposing the application.

1. Start with a *ubiquitous language*, a common vocabulary that is shared between all stakeholders.
2. Identify the relevant modules in the monolithic application and apply the common vocabulary to them.
3. Define the domain models of the monolithic application. The domain model is an abstract model of the business domain.

4. Define *bounded contexts* for the models. A bounded context is the boundary within a domain where a particular domain model applies. Apply explicit boundaries with clearly defined models and responsibilities.

The bounded contexts identified in step 4 are candidates for refactoring into smaller microservices. The following diagram shows the existing monolith with the bounded contexts overlaid:

For more information about using a DDD approach for microservices architectures, see Using domain analysis to model microservices.

Use glue code (anti-corruption layer)

While this investigative work is carried out to inventory the monolithic application, new functionality can be added by applying the principles of DDD as separate services. "Glue code" allows the monolithic application to proxy calls to the new service to obtain new functionality.

The glue code (adapter pattern) effectively acts as an anti-corruption layer, ensuring that the new service is not polluted by data models required by the monolithic application. The glue code helps to mediate interactions between the two and ensures that only data required by the new service is passed to enable compatibility.

Through the process of refactoring, teams can inventory the monolithic application and identify candidates for

microservices refactoring while also establishing new functionality with new services.

For more information about anti-corruption layers, see Anti-Corruption Layer pattern.

Create a presentation layer

The next step is to separate the presentation layer from the backend layer. In a traditional n-tier application, the application (business) layer tends to be the components that are core to the application and have domain logic within them. These coarse-grained APIs interact with the data access layer to retrieve persisted data from within a database. These APIs establish a natural boundary to the presentation tier, and help to decouple the presentation tier into a separate application space.

The follow diagram shows the presentation layer (UI) split out from the application logic and data access layers.

This diagram also introduces another layer, the API gateway, that sits between the presentation layer and the application logic. The API gateway is a façade layer that provides a consistent and uniform interface for the presentation layer to interact with, while allowing downstream services to evolve independently, without affecting the application. The API Gateway may use a technology such as Azure API Management, and allows the application to interact in a RESTful manner.

The presentation tier can be developed in any language or framework that the team has expertise in, such as a single page application or an MVC application. These applications interact with the microservices via the gateway, using standard HTTP calls. For more information about API Gateways, see Using API gateways in microservices.

Start to retire the monolith

At this stage, the team can begin peeling away the monolithic application and slowly extract the services that have been established by their bounded contexts into their own set of microservices. The microservices can expose a RESTful interface for the application layer to interact with, through the API gateway, with glue code in place to communicate with the monolith in specific circumstances.

As you continue to peel away the monolith, eventually there will come the point when it no longer needs to exist, and the microservices have been successfully extracted from the monolith. At this point, the anti-corruption layer (glue code) can safely be removed.

This approach is an example of the Strangler Fig pattern and allows for a controlled decomposition of a monolith into a set of microservices. Over time, as existing functionality is moved into microservices, the monolith will shrink in size and complexity, to the point that it no longer exists.

Next steps

When the application has been decomposed into constituent microservices, it becomes possible to use modern orchestration tools such as Azure DevOps to manage the lifecycle of each service. For more information, see CI/CD for microservices architectures.

Modernize enterprise applications with Azure Service Fabric

11/19/2021 • 28 minutes to read • Edit Online

This article provides guidelines for moving Windows applications to an Azure compute platform without rewriting. This migration uses container support in Azure Service Fabric.

A typical approach for migrating existing workloads to the cloud is the lift-and-shift strategy. In IaaS virtual machine (VM) migrations, you provision VMs with network and storage components and deploy the existing applications onto those VMs. Unfortunately, lift-and-shift often results in overprovisioning and overpaying for compute resources. Another approach is to move to PaaS platforms or refactor code into microservices and run in newer serverless platforms. But those options typically involve changing existing code.

Containers and container orchestration offer improvements. Containerizing an existing application enables it to run on a cluster with other applications. It provides tight control over resources, scaling, and shared monitoring and DevOps.

Optimizing and provisioning the right amount of compute resources for containerization isn't trivial. Service Fabric's orchestration allows an organization to migrate Windows and Linux applications to a runtime platform without changing code and to scale the needs of the application without overprovisioning VMs. The result is better density, better hardware use, simplified operations, and overall lower cloud-compute costs.

An enterprise can use Service Fabric as a platform to run a large set of existing Windows-based web applications with improved density, monitoring, consistency, and DevOps, all within a secure extended private network in the cloud. The principle is to use Docker and Service Fabric's containerization support that packages and hosts existing web applications on a shared cluster with shared monitoring and operations, in order to maximize cloud compute resources for the ideal performance-to-cost ratio.

This article describes the processes, capabilities, and Service Fabric features that enable containerizing in an optimal environment for a large enterprise. The guidance is scoped to web applications and Windows containers. Before reading this article, get familiar with core Windows container and Service Fabric concepts. For more information, see:

- Create your first Service Fabric container application on Windows
- Service Fabric terminology overview
- Service Fabric best practices overview

Resources

Sample: Modernization templates and scripts.

The repo has these resources:

- An example Azure Resource Manager template to bring up an Azure Service Fabric cluster.
- A reverse-proxy solution for brokering web requests into the Service Fabric cluster to the destination containers.
- Sample Service Fabric application configuration and scripts that show the use of placement, resource constraints, and autoscaling.
- Sample scripts and Docker files that build and package an existing web application.

Customize the templates in this repo for your cluster. The templates implement the best practices described in

this article.

Evaluate requirements

Before containerizing existing applications, evaluate requirements. Select applications that are right for migration, choose the right developer workstation, and determine network requirements.

Application selection

First, determine the type of applications that are best suited for a containerized platform, full virtual machines, and pure PaaS environment. The application could be a shared application that is built with Service Fabric to share Windows Server hosts across various containerized applications. Each Service Fabric host can run multiple different applications running in isolated Windows containers.

Consider creating a set of criteria to determine such applications. Here are some example criteria of containerized Windows applications in Service Fabric.

- HTTP/HTTPS web and application tiers without database dependency.
- Stateless web applications.
- Built with .NET Framework versions 3.5 and later.
- Do not have hardware dependency or access device drivers.
- Applications can run on Windows Server 2016 and later versions.
- All dependencies can be containerized, such as are most .NET assemblies, WCF, COM+.

  > **NOTE**
  > Dependencies that cannot be containerized include MSMQ (Currently supported in preview releases of Windows Server Core post 1709).

- Applications can compile and build in Visual Studio.

For the web applications, databases, and other required servers (such as Active Directory) exist outside the Service Fabric cluster in IaaS VMs, PaaS, or on-premises.

Developer workstation requirements

From an application development perspective, determine the workstation requirements.

- Docker for Windows is required for developers to containerize and test their applications prior to deployment.
- Visual Studio Docker support is required. Standardize on the latest version of Visual Studio for the best Docker compatibility.
- If workstations don't have enough hardware resources to oversee those requirements, use Azure compute resources for speed and productivity gains. An option is the Azure DevTest Labs Service. Docker for Windows, and Visual Studio 2019 require a minimum of 8 GB of memory.

Networking requirements

Service Fabric orchestration provides a platform for hosting, deploying, scaling, and operating applications at enterprise scale. Most large enterprises that use Azure:

- Extend their corporate network with a private address space to an Azure subscription. Use either ExpressRoute or a Site-to-Site VPN to provide secure on-premises connectivity.
- Want to control inbound and outbound network traffic through third-party firewall appliances and/or Azure Network Security Group rules.
- Want tight control over the address space requirements and subnets.

Service Fabric is suitable as a containerization platform. It plugs into an existing cloud infrastructure and doesn't require open public ingress endpoints. You just need to carve out the necessary address space for Service Fabric's IP address requirements. For details, see the Service Fabric Networking section in this article.

Containerize existing Windows applications

After you've determined the applications that meet the selection criteria, containerize them into Docker images. The result is containerized .NET web application running in IIS where all tiers run in one container.

> **NOTE**
>
> You can use multiple containers; one per tier.

Here are the basic steps for containerizing an application.

1. Open the project in Visual Studio.
2. Make sure the project compiles and runs locally on the developer workstation.
3. Add a Dockerfile to the project. This Dockerfile example shows a basic .NET MVC application.

```
FROM mcr.microsoft.com/dotnet/framework/aspnet:4.8
ADD PublishOutput/ /inetpub/wwwroot

# add a certificate and configure SSL
SHELL ["powershell","-command"]
RUN New-Item -ItemType Directory C:\install
ADD installwebsite.ps1 c:\install\
ADD wildcard.pfx c:\install\
# Powershell script to configure SSL on the website
RUN c:\install\configurewebsite

# plugin into SF healthcheck ensuring the container website is running
HEALTHCHECK --interval=30s --timeout=30s --start-period=60s --retries=3 CMD curl -f http://localhost/
|| exit 1
```

4. Test locally by using Docker For Windows. The application must successfully run in a Docker container by using the Visual Studio debug experience. For more information, see Deploy a .NET app using Docker Compose.
5. Build (if needed), tag, and push the tested image to a container registry, like the Azure Container Registry service. This example uses an existing Azure Container Registry named MyAcr and Docker build/tag/push to build/deploy appA to the registry.

```
docker login myacr.azurecr.io -u myacr -p <pwd>
docker build -t appa .
docker tag appa myacr.azurecr.io/appa:1.0
docker push myacr.azurecr.io/appa:1.0
```

The image is tagged with a version number that Service Fabric references when it deploys and versions the container. Azure DevOps encapsulates and executes the manual Docker build/tag/push process. DevOps details are described in the DevOps and CI/CD section.

> **NOTE**
> In the preceding example, the base image is "mcr.microsoft.com/dotnet/framework/aspnet:4.8" from the Microsoft Container Registry.

Here are some considerations about the base images:

- The base image could be a locked-down custom enterprise image that enforces enterprise requirements. For a shared application, isolation boundaries can be created through credentials or by using separate registry. It's recommended that enterprise-supported docker images be kept separately and stored in an isolated container registry.
- Avoid storing the registry login credentials in configuration files. Instead, use role-based access control (RBAC) and Azure Active Directory service principals with Azure Container Registry. Provide read-only access to registries depending on your enterprise requirements.

For information about running an IIS ASP.NET MVC application in a Windows container, see Migrating ASP.NET MVC Applications to Windows Containers.

Service Fabric cluster configuration for enterprise deployments

To deploy a Service Fabric cluster, start with the sample Azure Resource Manager template in this GitHub Repo and customize it to fit your requirements. You also deploy a cluster through the Azure portal, but that option should be used for development/test provisioning.

Service Fabric node types

A Service Fabric cluster contains one or more node types. A node type typically maps to an Azure virtual machine scale set which is a set of one or more VM instances with same properties. The cluster must have at least one node type-primary node type, which runs the Service Fabric system services. Do not run your application container on the primary node type because the container can compete with the system services for resources. Consider, designing a cluster with two or more node types depending on the types of applications. For example, if you have two node types, run HTTP web and application tier containers on a non-primary node type.

This example infrastructure uses two Service Fabric node types: application and primary. You can scale in or scale out the scale set individually. It doesn't require extensive planning and testing up front to determine the correct application node type size (scale set instance count), because the actual size can grow and reduce in real time driven by monitoring and metrics.

In the preceding image, the application node type has three nodes that run three containerized applications A, B, and C. The containers can scale on demand based on CPU and memory usage. The hosts can scale out as more applications are deployed to the cluster. Azure Load Balancer has a public IP. The primary nodes are in an internal subnet. It has a load balancer that isn't publicly exposed.

Placement constraints

Use placement constraints that target the non-primary node type to reserve the primary node type for system services.

This approach enables you to configure each scale set with a separate virtual network subnet. A unique subnet for each node type uses network security group rules for inbound/outbound access to and from the subnet and controls network flow. You can configure the primary node type with a private load balancer so that external traffic can't access Service Fabric management and application deployment. If the application node type wants to expose application endpoints publicly, configure it with a separate load balancer with security configuration. For an example template that uses Windows Service with NSG rule and multiple node types, see 7 Node, 3 node type secure Windows Service Fabric Cluster with NSG.

A scale set associated with a node type can reliably scale out to 100 VM instances by using a single placement group as applications are added to the cluster. The primary node type often doesn't need as many instances and can run with 5-7 nodes, depending on durability and reliability requirements.

Service Fabric networking

Service Fabric supports with two networking modes for containerized applications; nat and Open. For large enterprise clusters that host multiple applications, use the Open mode. For more information, see Container Networking and Constraints.

- NAT

  By default, the cluster brings up containers by using a NAT-bridge mode to the host VM. The NAT bridge routes requests over a defined port to the container. With this mode, only one IP address is needed per host VM for the host's primary NIC.

  To route traffic to each application container, a unique port is exposed through the load balancer. However, that port is exposed to end users. If you don't want the port exposed, provide a URL rewrite mechanism. Rewrite the application domain name with a unique application port. Traffic is routed to the load balancer that front ends the cluster. One option for the rewrite mechanism is Azure Application

Gateway.

Another benefit of this approach is simplistic load balancing with Azure Load Balancer. The load balancer's probe mechanism balance traffic across the VM instances that are running the application's containers.

- **Open**

  The **Open** mode assigns an IP address to each running container on the host VM from the cluster's virtual network subnet. Each host is pre-allocated with a set of IP addresses. Each container on the host is assigned an IP from the virtual network range. You can configure this mode in the cluster Azure Resource Manager template during cluster creation. The example infrastructure demonstrates the **Open** mode.

  Benefits of the Open mode:

  - Makes connecting to application containers simple.
  - Provides application traceability that is, the assigned enterprise-friendly IP is constant for the life of the container.
  - Is efficient with Windows containers.

There are downsides:

- The number of IP addresses must be set aside during cluster creation and that number is fixed. For example, if 10 IP addresses are assigned to each host, the host supports up to 9 application containers; one IP is reserved for the host VM's primary NIC, the remaining addresses for each container. The number of IP addresses to configure for each host is determined based on the hardware size for the node type, and the maximum number of containers on each host. Both factors depend on application size and needs. If you need more containers in the cluster, add more application node type VM instances. That is because you can't change the IP number per instance without rebuilding the cluster.
- You need a reverse proxy to route traffic to the correct destination container. The Service Fabric DNS Service can be used by the reverse proxy to look up the application container and rewrite the HTTP request to this container. That solution is more complex to implement than the **nat** mode.

For more information, see Service Fabric container networking modes.

Service Fabric runtime and Windows versions

Choose the version of Windows Server you want to run as the infrastructure host OS. The choice depends on the testing and readiness at the time of first deployment. Supported versions are Windows Server with Containers 1607, 1709, 1803.

Also choose a version of the Service Fabric runtime to start the cluster. For a new containerization initiative, use the latest version (Service Fabric 6.4 or later).

Other considerations:

- The host OS running Service Fabric can be built by using locked down enterprise images with antivirus and malware enforcement.
- The host OS for all the Service Fabric cluster VMs can be domain joined. Domain joining isn't a requirement and can add complexity to any Azure virtual machine. However, there are benefits. For example, if an application requires Windows-integrated security to connect to domain-joined resources, then the domain service account is typically used at the process level. The account is used to execute the application instead of connection string credentials and secrets. Windows containers do not currently support direct domain joining but can be configured to use Group Managed Storage Accounts (gMSA). The gMSA capability is supported by Service Fabric for Windows-integrated security requirements. Each containerized application in the cluster can have its own gMSA. gMSA requires the Service Fabric host VMs that run containers to be Active Directory domain joined.

DNS service

Service Fabric has an internal DNS service that maps a containerized application name to its location in the cluster. For the **Open** mode, the application's IP address is used instead. This service is enabled on the cluster. If you name each service with a DNS name, traffic is routed to the application by using a reverse proxy. For information about reverse proxy, see the Reverse proxy for inbound traffic section.

Monitoring and diagnostics

The Service Fabric Log Analytics workspace and Service Fabric solution provide detailed information about cluster events. For information about setting it up, see Configure Azure Monitor logs to collect cluster events.

- Install the [monitoring agent] for Azure Log Analytics in the Service Fabric cluster. You can then use the container monitoring solution and view the running containers in the cluster.
- Use Docker performance statistics to monitor memory and CPU use for each container.
- Install the Azure Diagnostics extension that collects diagnostic data on a deployed application. For more information, see What is Azure Diagnostics extension.

Unused container images

Container images are downloaded to each Service Fabric host and can consume space on the host disk. To free up disk space, consider image pruning to remove images that are no longer referenced and used by running containers. Configure this option in the Host section of the cluster manifest.

Secrets and certificates management with Key Vault

Avoid hardcoded secrets in the template. Modify the Azure Resource Manager template to add certificates (cluster certificate) and secrets (host OS admin password) from Azure Key Vault during cluster deployment. To safely store and retrieve secrets:

1. Create a Key Vault in a separate resource group in the same region as the cluster.
2. Protect the Key Vault with RBAC.
3. Load the cluster certificate and cluster password in Key Vault.
4. Reference this external Key Vault from the Azure Resource Manager template.

Cluster capacity planning

For an enterprise production cluster, start with 5-7 primary nodes, depending on the intended size of the overall cluster. For more information, see Service Fabric cluster capacity planning considerations.

For a large cluster that hosts stateless web applications, the typical hardware size for the primary node type can be Standard_D2s_v3. For the primary node type, a 5-node cluster with the Silver durability tier should scale to containers across 50 application VM instances. Consider doing simulation tests and add more node types as partitioning needs become apparent. Also perform tests to determine the VM size for application node type and the number of running containers on each VM instance. Typically, 10 or more containers run on each VM, but that number is dependent on these factors:

- Resource needs, such as CPU and memory use, of the containerized applications. Web applications tend to be more CPU intensive than memory intensive.
- The compute size and available resources of the chosen node VM for the application node type.
- The number of IP addresses configured per node. Each container requires an IP with Open mode. Given 15 containers on a VM, 16 IP addresses must be allocated for each VM in the application node type.
- Resource needs of the underline OS. The recommendation is to leave 25% of the resources on the VM for OS system processes.

Here are some recommendations: Choose recent Windows Server builds because older builds of Windows Server (1607) use larger container image sizes compared to newer versions (1709 and later).

Select the most appropriate VM compute size and initial application node type VM count. Standard_D8s_v3,

Standard_D16s_v3, and Standard_DSv32_v3 are example compute sizes that are known to work well for this tier for running containerized web applications.

Use premium storage for the production cluster because container images must be read quickly from disk. Attach one data disk to each VM and move the Docker EE installation to that disk. That way there is enough log space for each running container. A standard VM with the default 127-GB OS disk fills up because of container logs. A 500 GB to 1 TB data disk is recommended per VM for the application node type scale set.

Availability and resiliency planning

Service Fabric spreads VM instances across fault and update domains to ensure maximum availability and resilience. For an enterprise-scale production cluster, as a minimum target the Silver tier (5-node cluster) for reliability and durability when hosting containerized stateless web applications.

Scale planning

There are two aspects to consider:

- Scale in or scale out the scale set associated with the application node type by adding or removing VM instances. VMs can be added and removed automatically through autoscaling or manually through scripts. For an enterprise cluster, do not add or remove nodes frequently. It is important that you disable nodes in Service Fabric before deleting them. Do not delete seed nodes from your cluster. Monitor the cluster VMs with alerts to trigger when VM resources exceed threshold values.
- Scale the application's container count across the scale set instances in the application node type.

For the application node type, start with the minimal required nodes to support the containerized applications and ensure high availability. Have extra nodes in the cluster. A node can be removed from the cluster for maintenance and its running containers can be temporarily moved to other nodes. The cluster can grow statically using the `Add-AzureRmServiceFabricNode` cmdlet, or dynamically by using scale set autoscale capability.

Starting with Service Fabric 6.2, application containers can autoscale individually through configuration. The configuration is based on Docker statistics for CPU and memory use on the host. Together, those capabilities can optimize the compute cost.

- Rather than overprovisioning a VM to run an application, deploy a container to the cluster, monitor it, and then scale out with additional containers, as necessary. Choosing the right size is easier because Docker statistics are used to determine the number of containers.

In the example infrastructure, application A has two containers deployed across the cluster that divide load. This approach allows the application and container combination to be adjusted later for optimization.

> **NOTE**
>
> Docker statistics showing individual container resource utilization is sent to Log Analytics and can be analyzed in Azure Monitor.

- Service Fabric offers constant monitoring and heath checks across the cluster. If a node is unhealthy, applications on that node automatically move to a healthy node and the bad node stops receiving requests. Regardless of the number of containers hosting an application, Service Fabric ensures that the application is healthy and running.

For an application that is infrequently used and can be offline, run it in the cluster with just one container instance (such as application B and C). Service Fabric makes sure that the application is up and healthy during upgrades or when the container needs to move to a new VM. Heath checking can reduce cost compared to hosting that application on two redundant and overprovisioned VMs in the traditional IaaS model.

Container networking and constraints

Use the **Open** mode for hosting containerized web applications in the cluster. After deployment, the application is immediately discoverable through the Service Fabric DNS service. The DNS service is a name-value lookup between a configured service name and the resultant IP address of a container hosting the application.

To route web requests to an application, use an ingress reverse proxy. If application containers listen on different ports (AppA port 8080, AppB on 8081), the default host NAT bridge works without issues. Azure Load Balancer probes route the traffic appropriately. However, if you want incoming traffic over SSL/443 routed to one port for all applications, use a reverse proxy to route traffic appropriately.

Reverse proxy for inbound traffic

Service Fabric has a built-in reverse proxy but is limited in its feature set. Therefore, deploy a different reverse proxy. An option is the IIS Application Request Routing (ARR) extension for IIS hosted web applications. The ARR can be deployed to a container and configured to take inbound requests and route them to the appropriate application container. In this example, the ARR uses a NAT bridge over port 80/443, accepts all inbound web traffic, inspects the traffic, looks up the destination container using Service Fabric DNS service, and rewrites the request to the destination container. The traffic can be secured with SSL to the destination container. Follow the IIS Application Request Routing sample for building an ARR reverse proxy. For information, see Using the Application Request Routing Module.

Here is the network flow for the example infrastructure.

The key aspect of the ingress reverse proxy is inspecting inbound traffic and rewriting that traffic to the destination container.

For example, application A is registered with the Service Fabric DNS service with the domain name: appA.container.myorg.com. External users access the application with `https://appA.myorg.com`. Use public or organizational DNS and register appA.myorg.com to point to the public IP for the application node type.

1. Requests for appA.myorg.com are routed to the Service Fabric cluster and handed off to the ARR container listening on port 443. Service Fabric and Azure Load Balancer set that configuration value when the ARR container is deployed.
2. When ARR gets the request, it has a condition to look for any request with the pattern='..*', and its action rewrites the request to `https://{C:1}.container.{C:2}.{C:3}/{REQUEST_URI}`. Because the ARR is running in the cluster, the Service Fabric DNS service is invoked. The service returns the destination container IP address.
3. The request is routed to the destination container. Certificates can be used for the initial request to ARR and the rewrite to the destination container.

Here is an example ApplicationManifest.xml for Container A in the example infrastructure.

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationManifest ApplicationTypeName="sfapp02Type"
                     ApplicationTypeVersion="1.0.0"
                     xmlns="http://schemas.microsoft.com/2011/01/fabric"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Parameters>
    <Parameter Name="appAsvc_InstanceCount" DefaultValue="2" />
  </Parameters>
  <ServiceManifestImport>
    <ServiceManifestRef ServiceManifestName="appAsvcPkg" ServiceManifestVersion="1.0.0" />
    <ConfigOverrides />
    <Policies>
      <ContainerHostPolicies CodePackageRef="Code" Isolation="default">
        <!--SecurityOption Value="credentialspec=file://container_gmsa1.json"/-->
        <RepositoryCredentials AccountName="myacr" Password=" " PasswordEncrypted="false"/>
        <NetworkConfig NetworkType="Open"/>
      </ContainerHostPolicies>
      <ServicePackageResourceGovernancePolicy CpuCores="1" MemoryInMB="1024" />
    </Policies>
  </ServiceManifestImport>
  <DefaultServices>
    <Service Name="appAsvc" ServicePackageActivationMode="ExclusiveProcess"
ServiceDnsName="appA.container.myorg.com">
      <StatelessService ServiceTypeName="appAsvcType" InstanceCount="[appAsvc_InstanceCount]">
        <SingletonPartition />
        <PlacementConstraints>(NodeTypeName==applicationNT)</PlacementConstraints>
      </StatelessService>
    </Service>
  </DefaultServices>
</ApplicationManifest>
```

- Uses the **Open** mode for the containers.
- Registers the application domain name **appA.container.myorg.com** with the Azure DNS service.
- Optionally configures the container to use an Active Directory gMSA (commented).
- Uses placement constraints to deploy the container to the application node type, named applicationNT. It instructs Service Fabric to run the container in the correct node type in the secured network subnet.
- Optionally applies resource constraints. Each container is resource governed to use 1 vCPU and 1 GB of memory on the VM host. Setting a resource governance policy is recommended because Service Fabric uses the policy to distribute containers across the cluster, as opposed to the default even distribution of containers across the cluster.

Here is the example ServiceManifest.xml for the containerized application appA.

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceManifest Name="appAsvcPkg"
                 Version="1.0.0"
                 xmlns="http://schemas.microsoft.com/2011/01/fabric"
                 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ServiceTypes>
    <StatelessServiceType ServiceTypeName="appAsvcType" UseImplicitHost="true" />
  </ServiceTypes>
  <CodePackage Name="Code" Version="1.0.0">
    <EntryPoint>
      <ContainerHost>
        <ImageName>myacr.azurecr.io/appa:1.0</ImageName>
      </ContainerHost>
    </EntryPoint>
  </CodePackage>
  <ConfigPackage Name="Config" Version="1.0.0" />
  <Resources>
    <Endpoints>
      <Endpoint Name="appAsvcTypeEndpoint1" UriScheme="http" Port="80" Protocol="http"
CodePackageRef="Code"/>
      <Endpoint Name="appAsvcTypeEndpoint2" UriScheme="https" Port="443" Protocol="http"
CodePackageRef="Code"/>
    </Endpoints>
  </Resources>
</ServiceManifest>
```

- The application is configured to listen on ports 80 and 443. By using the **Open** mode and reverse proxy, all applications can share the same ports.
- The application is containerized to an image named: myacr.azurecr.io/appa:1.0. Service Fabric invokes the Docker daemon to pull down the image when the application is deployed. Service Fabric handles all interactions with Docker.

The reverse proxy container uses similar manifests but isn't configured to use the **Open** mode. You can update containers by versioning the Docker image, then redeploying the versioned Service Fabric package with the **Start-ServiceFabricApplicationUpgrade** cmdlet.

For information about manifests, see Service Fabric application and service manifests.

Environmental configuration

Do not hardcode configuration values in the container image by using environment variables to pass values to a container. A DevOps pipeline can build a container image, test in a test environment, promote to staging (or pre-production), and promote to production. Do not rebuild an image for each environment.

Docker can pass environment variables directly to a container when starting one. In this example, Docker passes the eShopTitle variable to the eshopweb container:

```
docker run -p 80:80 -d --name eshoptest -e eShopTitle=SomeName eshopweb:1.0
```

In a Service Fabric cluster, Service Fabric controls Docker execution and lists environment variables in the ServiceManifest. Those variables are passed automatically when Service Fabric runs the container. You can override the variables in ApplicationManifest.xml by using the *EnvironmentOverrides* element, which can be parameterized and built from Visual Studio publish profiles for each environment.

For information about specifying environment variables in, see How to specify environment variables for services in Service Fabric.

Security considerations

Here are some articles about container security:

Azure Service Fabric security

Service Fabric application and service security

Set up an encryption certificate and encrypt secrets on Windows clusters

Logging and monitoring

Monitoring and logging are critical to operational success and is achieved through integration with Azure Monitor and Log Analytics.

Monitor the Service Fabric cluster and each executing containers by using the scale set extension agent for Log Analytics and its associated Container Monitoring Solution. Make sure that you configure and install the extension agent and solution during cluster creation. Docker statistics for container CPU and memory utilization are sent to Log Analytics and can be queried for proactive monitoring and alerting. Set up proactive alerts through Azure Monitor. Here are metrics that you should monitor.

- High CPU and memory utilization of a container.
- Container count hitting a per VM threshold.
- HTTP error codes and count.
- Docker Enterprise Edition (Docker EE) up or down on the host VM.
- Response time of a service.
- Host VM-based alerts on CPU, memory, disk, file consumption.

These Service Fabric virtual machine scale set extensions are installed on a typical node that results in logging.

- ServiceFabricNode. Links the node to a storage account (support log) for tracing support. This log is used when a ticket is opened.
- IaaSDiagnostics. Collects platform events, such as ServiceFabricSystemEventTable, and stores that data in a blob storage account (app log). The account is consumed in Log Analytics.
- MicrosoftMonitoringAgent. Contains all the performance data such as Docker statistics. The data (such as ContainerInventory and ContainerLog) is sent to Log Analytics.

Application log

If your containerized application runs in a shared cluster, you can get logs such as IIS and custom logs from the container into Log Analytics. This option is recommended because of speed, scalability, and the ability to handle large amounts of unstructured data.

Set up log rotation through Docker to keep the logs size manageable. For more information, see Rotating Docker Logs - Keeping your overlay folder small.

Here are two approaches for getting application logs into Log Analytics.

- Use the existing Container Monitoring Solution installed in Log Analytics. The solution automatically sends data from the container log directories on the VM (C:\ProgramData\docker\windowsfilter*) to the configured Log Analytics workspace. Each container creates a directory underneath the \WindowsFilter path, and the contents are streamed to Log Analytics from MicrosoftMonitoringAgent on the VM. This way you can send application logs to a shared directory(s) in the container and relay the logs by using Docker Logs to the monitored container log folders.

  1. Write a process script that runs in the container periodically and analyzes log files.
  2. Monitor the log file changes in the shared folder and write the log changes to the command window where Docker Logs can capture the information outside the container.

  Each container records any output sent to the command line of a container. Access the output outside the container, which is automatically executed by Container Monitoring Solution.

  ```
  docker logs <ContainerID>
  ```

  Move Docker to an attached VM data disk with enough storage to make sure the OS drive doesn't fill up with container logs.

  The automatic Container Monitoring Solution sends all logs to a single Log Analytics workspace. Different containerized applications running on the same host send application logs to that shared workspace. If you need to isolate logs such that each containerized application sends the log to a separate workspace, supply a custom solution. That content is outside the scope of this article.

- Mount external storage to each running container by using a file management service such as Azure Files. The container logs are sent to the external storage location and don't take up disk space on the host VM.

  - You don't need, a data disk attached to each VM to hold Docker logs; move Docker Enterprise to the data disk.
  - Create a job to monitor the Azure Files location and send logs to the appropriate Log Analytics workspace for each installed application. The job doesn't need to run in the container. It just observes the Azure Files location.

DevOps and CI/CD

Application containerization ensures consistency. It makes sure all Service Fabric-hosted applications use the latest approved corporate image and provides an automatable image updating process that is consistent through DevOps. Azure Pipelines provides the automation process. For more information, see Tutorial: Deploy an application with CI/CD to a Service Fabric cluster.

- An enterprise may want to control the base container images in a centralized registry. The preceding workflow shows one image registry. There could be multiple registries that are used to share central IT-built enterprise images with application teams. One way to centralize control is for the central IT registry to allow application teams with read-only access to the enterprise base image repository. Application teams each have their own container registry with their Docker files and build off the central IT base image repository.
- There are various third-party image scanning tools that can plug into this process on push/pulls from the Azure Container Registry. Those solutions are available in Azure Marketplace and referenced in the Azure portal Container Registry blade. For example, Aqua and Prisma Cloud.
- After the source code is pushed to a git-based repository, set up CI/CD by creating an Azure DevOps build definition, selecting the source repository, and choosing the **Azure Service Fabric Application and Docker Support** template.

The template sets up the build process and tasks for CI/CD by building and containerizing the application, pushing the container image to a container registry (Azure Container Registry is the default), and deploying the

Service Fabric application with the containerized services to the cluster. Each application code change creates a version of the code and an updated containerized image. Service Fabric's rolling upgrade feature deploys service upgrades gracefully.

Here is an example of a build starting the full DevOps process on an Azure-provided hosted build agent. Some enterprises may require the build agents to run internally within their private Azure virtual network corporate network. Set up a Windows build agent VM and instruct Azure DevOps to use the private VM for building and deploying code. For information about using custom build agents, see Self-hosted Windows agents.

Conclusion

Here is the summary of best practices:

- Before containerizing existing applications, selecting applications that are suitable for this migration, choose the right developer workstation, and determine network requirements.
- Do not run your application container on the primary node type. Instead, configure the cluster with two or more node types and run the application tier containers on a non-primary node type. Use placement constraints that target the non-primary node type to reserve the primary node type for system services.
- Use the **Open** networking mode.
- Use an ingress reverse proxy, such as the IIS Application Request Routing. The reverse proxy inspects inbound traffic and rewrites the traffic to the destination container.
- Do not hardcode configuration values in the container image and use environment variables to pass values to a container.
- Monitor the application, platform events, and infrastructure metrics by using IaasDiagnostics and MicrosoftMonitoringAgent extensions. View the logs in Application Insights and Log Analytics.
- Use the latest approved corporate image and provide an automatable image updating process that is consistent through DevOps.

Next steps

Get the latest version of the tools you need for containerizing, such as Visual Studio and Docker for Windows.

Customize these templates to meet your requirements. Sample: Modernization templates and scripts.

Migrate an Azure Cloud Services application to Azure Service Fabric

11/19/2021 • 18 minutes to read • Edit Online

Sample code

This article describes migrating an application from Azure Cloud Services to Azure Service Fabric. It focuses on architectural decisions and recommended practices.

For this project, we started with a Cloud Services application called Surveys and ported it to Service Fabric. The goal was to migrate the application with as few changes as possible. Later in the article, we show how to optimize the application for Service Fabric.

Before reading this article, it will be useful to understand the basics of Service Fabric. See Overview of Azure Service Fabric

About the Surveys application

A fictional company named Tailspin created an application called the Surveys app that allows customers to create surveys. After a customer signs up for the application, users can create and publish surveys, and collect the results for analysis. The application includes a public website where people can take a survey. Read more about the original Tailspin scenario here.

Now Tailspin wants to move the Surveys application to a microservices architecture, using Service Fabric running on Azure. Because the application is already deployed as a Cloud Services application, Tailspin adopts a multi-phase approach:

1. Port the cloud services to Service Fabric, while minimizing changes to the application.
2. Optimize the application for Service Fabric, by moving to a microservices architecture.

In a real-world project, it's likely that both stages would overlap. While porting to Service Fabric, you would also start to rearchitect the application into micro-services. Later you might refine the architecture further, perhaps dividing coarse-grained services into smaller services.

The application code is available on GitHub. This repo contains both the Cloud Services application and the Service Fabric version.

Why Service Fabric?

Service Fabric is a good fit for this project, because most of the features needed in a distributed system are built into Service Fabric, including:

- **Cluster management**. Service Fabric automatically handles node failover, health monitoring, and other cluster management functions.
- **Horizontal scaling**. When you add nodes to a Service Fabric cluster, the application automatically scales, as services are distributed across the new nodes.
- **Service discovery**. Service Fabric provides a discovery service that can resolve the endpoint for a named service.
- **Stateless and stateful services**. Stateful services use reliable collections, which can take the place of a cache or queue, and can be partitioned.

- **Application lifecycle management**. Services can be upgraded independently and without application downtime.
- **Service orchestration** across a cluster of machines.
- **Higher density** for optimizing resource consumption. A single node can host multiple services.

Service Fabric is used by various Microsoft services, including Azure SQL Database, Cosmos DB, Azure Event Hubs, and others, making it a proven platform for building distributed cloud applications.

Comparing Cloud Services with Service Fabric

The following table summarizes some of the important differences between Cloud Services and Service Fabric applications. For a more in-depth discussion, see Learn about the differences between Cloud Services and Service Fabric before migrating applications.

AREA	CLOUD SERVICES	SERVICE FABRIC
Application composition	Roles	Services
Density	One role instance per VM	Multiple services in a single node
Minimum number of nodes	2 per role	5 per cluster, for production deployments
State management	Stateless	Stateless or stateful*
Hosting	Azure	Cloud or on-premises
Web hosting	IIS**	Self-hosting
Deployment model	Classic deployment model	Resource Manager
Packaging	Cloud service package files (.cspkg)	Application and service packages
Application update	VIP swap or rolling update	Rolling update
Autoscaling	Built-in service	Virtual machine scale sets for auto scale out
Debugging	Local emulator	Local cluster

* Stateful services use reliable collections to store state across replicas, so that all reads are local to the nodes in the cluster. Writes are replicated across nodes for reliability. Stateless services can have external state, using a database or other external storage.

** Worker roles can also self-host ASP.NET Web API using OWIN.

The Surveys application on Cloud Services

The following diagram shows the architecture of the Surveys application running on Cloud Services.

The application consists of two web roles and a worker role.

- The **Tailspin.Web** web role hosts an ASP.NET website that Tailspin customers use to create and manage surveys. Customers also use this website to sign up for the application and manage their subscriptions. Finally, Tailspin administrators can use it to see the list of tenants and manage tenant data.
- The **Tailspin.Web.Survey.Public** web role hosts an ASP.NET website where people can take the surveys that Tailspin customers publish.
- The **Tailspin.Workers.Survey** worker role does background processing. The web roles put work items onto a queue, and the worker role processes the items. Two background tasks are defined: Exporting survey answers to Azure SQL Database, and calculating statistics for survey answers.

In addition to Cloud Services, the Surveys application uses some other Azure services:

- **Azure Storage** to store surveys, surveys answers, and tenant information.
- **Azure Cache for Redis** to cache some of the data that is stored in Azure Storage, for faster read access.
- **Azure Active Directory** (Azure AD) to authenticate customers and Tailspin administrators.
- **Azure SQL Database** to store the survey answers for analysis.

Moving to Service Fabric

As mentioned, the goal of this phase was migrating to Service Fabric with the minimum necessary changes. To that end, we created stateless services corresponding to each cloud service role in the original application:

Intentionally, this architecture is similar to the original application. However, the diagram hides some important differences. In the rest of this article, we'll explore those differences.

Converting the cloud service roles to services

For the initial migration, Tailspin followed the steps outlined in Guide to converting Web and Worker Roles to Service Fabric stateless services.

Creating the web front-end services

In Service Fabric, a service runs inside a process created by the Service Fabric runtime. For a web front end, that means the service is not running inside IIS. Instead, the service must host a web server. This approach is called *self-hosting*, because the code that runs inside the process acts as the web server host.

The original Surveys application uses ASP.NET MVC. Because ASP.NET MVC cannot be self-hosted in Service Fabric, Tailspin considered the following migration options:

- Port the web roles to ASP.NET Core, which can be self-hosted.
- Convert the web site into a single-page application (SPA) that calls a web API implemented using ASP.NET Web API. This would have required a complete redesign of the web front end.
- Keep the existing ASP.NET MVC code and deploy IIS in a Windows Server container to Service Fabric. This approach would require little or no code change.

The first option, porting to ASP.NET Core, allowed Tailspin to take advantage of the latest features in ASP.NET Core. To do the conversion, Tailspin followed the steps described in Migrating From ASP.NET MVC to ASP.NET Core MVC.

> **NOTE**
> When using ASP.NET Core with Kestrel, you should place a reverse proxy in front of Kestrel to handle traffic from the Internet, for security reasons. For more information, see Kestrel web server implementation in ASP.NET Core. The section Deploying the application describes a recommended Azure deployment.

HTTP listeners

In Cloud Services, a web or worker role exposes an HTTP endpoint by declaring it in the service definition file. A web role must have at least one endpoint.

```
<!-- Cloud service endpoint -->
<Endpoints>
    <InputEndpoint name="HttpIn" protocol="http" port="80" />
</Endpoints>
```

Similarly, Service Fabric endpoints are declared in a service manifest:

```
<!-- Service Fabric endpoint -->
<Endpoints>
    <Endpoint Protocol="http" Name="ServiceEndpoint" Type="Input" Port="8002" />
</Endpoints>
```

Unlike a cloud service role, Service Fabric services can be colocated within the same node. Therefore, every service must listen on a distinct port. Later in this article, we'll discuss how client requests on port 80 or port 443 get routed to the correct port for the service.

A service must explicitly create listeners for each endpoint. The reason is that Service Fabric is agnostic about communication stacks. For more information, see Build a web service front end for your application using ASP.NET Core.

Packaging and configuration

A cloud service contains the following configuration and package files:

FILE	DESCRIPTION
Service definition (.csdef)	Settings used by Azure to configure the cloud service. Defines the roles, endpoints, startup tasks, and the names of configuration settings.
Service configuration (.cscfg)	Per-deployment settings, including the number of role instances, endpoint port numbers, and the values of configuration settings.
Service package (.cspkg)	Contains the application code and configurations, and the service definition file.

There is one .csdef file for the entire application. You can have multiple .cscfg files for different environments, such as local, test, or production. When the service is running, you can update the .cscfg but not the .csdef. For more information, see What is the Cloud Service model and how do I package it?

Service Fabric has a similar division between a service *definition* and service *settings*, but the structure is more granular. To understand Service Fabric's configuration model, it helps to understand how a Service Fabric application is packaged. Here is the structure:

```
Application package
  - Service packages
    - Code package
    - Configuration package
    - Data package (optional)
```

The application package is what you deploy. It contains one or more service packages. A service package contains code, configuration, and data packages. The code package contains the binaries for the services, and the configuration package contains configuration settings. This model allows you to upgrade individual services without redeploying the entire application. It also lets you update just the configuration settings, without

redeploying the code or restarting the service.

A Service Fabric application contains the following configuration files:

FILE	LOCATION	DESCRIPTION
ApplicationManifest.xml	Application package	Defines the services that compose the application.
ServiceManifest.xml	Service package	Describes one or more services.
Settings.xml	Configuration package	Contains configuration settings for the services defined in the service package.

For more information, see Model an application in Service Fabric.

To support different configuration settings for multiple environments, use the following approach, described in Manage application parameters for multiple environments:

1. Define the setting in the Setting.xml file for the service.
2. In the application manifest, define an override for the setting.
3. Put environment-specific settings into application parameter files.

Deploying the application

Whereas Azure Cloud Services is a managed service, Service Fabric is a runtime. You can create Service Fabric clusters in many environments, including Azure and on premises. The following diagram shows a recommended deployment for Azure:

The Service Fabric cluster is deployed to a virtual machine scale set. Scale sets are an Azure Compute resource that can be used to deploy and manage a set of identical VMs.

As mentioned, it's recommended to place the Kestrel web server behind a reverse proxy for security reasons. This diagram shows Azure Application Gateway, which is an Azure service that offers various layer 7 load-balancing capabilities. It acts as a reverse-proxy service, terminating the client connection and forwarding requests to back-end endpoints. You might use a different reverse proxy solution, such as nginx.

Layer 7 routing

In the original Surveys application, one web role listened on port 80, and the other web role listened on port 443.

PUBLIC SITE	SURVEY MANAGEMENT SITE
`http://tailspin.cloudapp.net`	`https://tailspin.cloudapp.net`

Another option is to use layer 7 routing. In this approach, different URL paths get routed to different port numbers on the back end. For example, the public site might use URL paths starting with `/public/`.

Options for layer 7 routing include:

- Use Application Gateway.
- Use a network virtual appliance (NVA), such as nginx.
- Write a custom gateway as a stateless service.

Consider this approach if you have two or more services with public HTTP endpoints, but want them to appear as one site with a single domain name.

> One approach that we *don't* recommend is allowing external clients to send requests through the Service Fabric reverse proxy. Although this is possible, the reverse proxy is intended for service-to-service communication. Opening it to external clients exposes *any* service running in the cluster that has an HTTP endpoint.

Node types and placement constraints

In the deployment shown above, all the services run on all the nodes. However, you can also group services, so that certain services run only on particular nodes within the cluster. Reasons to use this approach include:

- Run some services on different VM types. For example, some services might be compute-intensive or require GPUs. You can have a mix of VM types in your Service Fabric cluster.
- Isolate front-end services from back-end services, for security reasons. All the front-end services will run on one set of nodes, and the back-end services will run on different nodes in the same cluster.
- Different scale requirements. Some services might need to run on more nodes than other services. For example, if you define front-end nodes and back-end nodes, each set can be scaled independently.

The following diagram shows a cluster that separates front-end and back-end services:

To implement this approach:

1. When you create the cluster, define two or more node types.
2. For each service, use placement constraints to assign the service to a node type.

When you deploy to Azure, each node type is deployed to a separate virtual machine scale set. The Service Fabric cluster spans all node types. For more information, see The relationship between Service Fabric node types and virtual machine scale sets.

> If a cluster has multiple node types, one node type is designated as the *primary* node type. Service Fabric runtime services, such as the Cluster Management Service, run on the primary node type. Provision at least 5 nodes for the primary node type in a production environment. The other node type should have at least 2 nodes.

Configuring and managing the cluster

Clusters must be secured to prevent unauthorized users from connecting to your cluster. It is recommended to use Azure AD to authenticate clients, and X.509 certificates for node-to-node security. For more information, see Service Fabric cluster security scenarios.

To configure a public HTTPS endpoint, see Specify resources in a service manifest.

You can scale out the application by adding VMs to the cluster. Virtual machine scale sets support autoscaling using autoscale rules based on performance counters. For more information, see Scale a Service Fabric cluster in or out using autoscale rules.

While the cluster is running, collect logs from all the nodes in a central location. For more information, see Collect logs by using Azure Diagnostics.

Refactor the application

After the application is ported to Service Fabric, the next step is to refactor it to a more granular architecture. Tailspin's motivation for refactoring is to make it easier to develop, build, and deploy the Surveys application. By decomposing the existing web and worker roles to a more granular architecture, Tailspin wants to remove the existing tightly coupled communication and data dependencies between these roles.

Tailspin sees other benefits in moving the Surveys application to a more granular architecture:

- Each service can be packaged into independent projects with a scope small enough to be managed by a small team.
- Each service can be independently versioned and deployed.
- Each service can be implemented using the best technology for that service. For example, a service fabric cluster can include services built using different versions of the .Net Frameworks, Java, or other languages such as C or C++.
- Each service can be independently scaled to respond to increases and decreases in load.

The source code for the refactored version of the app is available on GitHub.

Design considerations

The following diagram shows the architecture of the Surveys application refactored to a more granular architecture:

Tailspin.Web is a stateless service self-hosting an ASP.NET MVC application that Tailspin customers visit to create surveys and view survey results. This service shares most of its code with the *Tailspin.Web* service from the ported Service Fabric application. As mentioned earlier, this service uses ASP.NET core and switches from using Kestrel as web frontend to implementing a WebListener.

Tailspin.Web.Survey.Public is a stateless service also self-hosting an ASP.NET MVC site. Users visit this site to select surveys from a list and then fill them out. This service shares most of its code with the *Tailspin.Web.Survey.Public* service from the ported Service Fabric application. This service also uses ASP.NET Core and also switches from using Kestrel as web frontend to implementing a WebListener.

Tailspin.SurveyResponseService is a stateful service that stores survey answers in Azure Blob Storage. It also merges answers into the survey analysis data. The service is implemented as a stateful service because it uses a ReliableConcurrentQueue to process survey answers in batches. This functionality was originally implemented in the *Tailspin.AnswerAnalysisService* service in the ported Service Fabric application.

Tailspin.SurveyManagementService is a stateless service that stores and retrieves surveys and survey questions. The service uses Azure Blob storage. This functionality was also originally implemented in the data access components of the *Tailspin.Web* and *Tailspin.Web.Survey.Public* services in the ported Service Fabric application. Tailspin refactored the original functionality into this service to allow it to scale independently.

Tailspin.SurveyAnswerService is a stateless service that retrieves survey answers and survey analysis. The service also uses Azure Blob storage. This functionality was also originally implemented in the data access components of the *Tailspin.Web* service in the ported Service Fabric application. Tailspin refactored the original functionality into this service because it expects less load and wants to use fewer instances to conserve resources.

Tailspin.SurveyAnalysisService is a stateless service that persists survey answer summary data in a Redis cache for quick retrieval. This service is called by the *Tailspin.SurveyResponseService* each time a survey is answered and the new survey answer data is merged in the summary data. This service includes the functionality originally implemented in the *Tailspin.AnswerAnalysisService* service from the ported Service Fabric application.

Stateless versus stateful services

Azure Service Fabric supports the following programming models:

- The guest executable model allows any executable to be packaged as a service and deployed to a Service Fabric cluster. Service Fabric orchestrates and manages execution of the guest executable.

- The container model allows for deployment of services in container images. Service Fabric supports creation and management of containers on top of Linux kernel containers as well as Windows Server containers.
- The reliable services programming model allows for the creation of stateless or stateful services that integrate with all Service Fabric platform features. Stateful services allow for replicated state to be stored in the Service Fabric cluster. Stateless services do not.
- The reliable actors programming model allows for the creation of services that implement the virtual actor pattern.

All the services in the Surveys application are stateless reliable services, except for the *Tailspin.SurveyResponseService* service. This service implements a ReliableConcurrentQueue to process survey answers when they are received. Responses in the ReliableConcurrentQueue are saved into Azure Blob Storage and passed to the *Tailspin.SurveyAnalysisService* for analysis. Tailspin chooses a ReliableConcurrentQueue because responses do not require strict first-in-first-out (FIFO) ordering provided by a queue such as Azure Service Bus. A ReliableConcurrentQueue is also designed to deliver high throughput and low latency for queue and dequeue operations.

Operations to persist dequeued items from a ReliableConcurrentQueue should ideally be idempotent. If an exception is thrown during the processing of an item from the queue, the same item may be processed more than once. In the Surveys application, the operation to merge survey answers to the *Tailspin.SurveyAnalysisService* is not idempotent because Tailspin decided that the survey analysis data is only a current snapshot of the analysis data and does not need to be consistent. The survey answers saved to Azure Blob Storage are eventually consistent, so the survey final analysis can always be recalculated correctly from this data.

Communication framework

Each service in the Surveys application communicates using a RESTful web API. RESTful APIs offer the following benefits:

- Ease of use: each service is built using ASP.NET Core MVC, which natively supports the creation of Web APIs.
- Security: While each service does not require SSL, Tailspin could require each service to do so.
- Versioning: clients can be written and tested against a specific version of a web API.

Services in the Survey application use the reverse proxy implemented by Service Fabric. Reverse proxy is a service that runs on each node in the Service Fabric cluster and provides endpoint resolution, automatic retry, and handles other types of connection failures. To use the reverse proxy, each RESTful API call to a specific service is made using a predefined reverse proxy port. For example, if the reverse proxy port has been set to **19081**, a call to the *Tailspin.SurveyAnswerService* can be made as follows:

```
static SurveyAnswerService()
{
    httpClient = new HttpClient
    {
        BaseAddress = new Uri("http://localhost:19081/Tailspin/SurveyAnswerService/")
    };
}
```

To enable reverse proxy, specify a reverse proxy port during creation of the Service Fabric cluster. For more information, see reverse proxy in Azure Service Fabric.

Performance considerations

Tailspin created the ASP.NET Core services for *Tailspin.Web* and *Tailspin.Web.Surveys.Public* using Visual Studio templates. By default, these templates include logging to the console. Logging to the console may be done during development and debugging, but all logging to the console should be removed when the application is

deployed to production.

> **NOTE**
> For more information about setting up monitoring and diagnostics for Service Fabric applications running in production, see monitoring and diagnostics for Azure Service Fabric.

For example, the following lines in *startup.cs* for each of the web front end services should be commented out:

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    //loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    //loggerFactory.AddDebug();

    app.UseMvc();
}
```

> **NOTE**
> These lines may be conditionally excluded when Visual Studio is set to **release** when publishing.

Finally, when Tailspin deploys the Tailspin application to production, they switch Visual Studio to **release** mode.

Deployment considerations

The refactored Surveys application is composed of five stateless services and one stateful service, so cluster planning is limited to determining the correct VM size and number of nodes. In the *applicationmanifest.xml* file that describes the cluster, Tailspin sets the *InstanceCount* attribute of the *StatelessService* tag to -1 for each of the services. A value of -1 directs Service Fabric to create an instance of the service on each node in the cluster.

> **NOTE**
> Stateful services require the additional step of planning the correct number of partitions and replicas for their data.

Tailspin deploys the cluster using the Azure portal. The Service Fabric Cluster resource type deploys all of the necessary infrastructure, including virtual machine scale sets and a load balancer. The recommended VM sizes are displayed in the Azure portal during the provisioning process for the Service Fabric cluster. Because the VMs are deployed in a virtual machine scale set, they can be both scaled up and out as user load increases.

Next steps

The Surveys application code is available on GitHub.

If you are just getting started with Azure Service Fabric, first set up your development environment then download the latest Azure SDK and the Azure Service Fabric SDK. The SDK includes the OneBox cluster manager so you can deploy and test the Surveys application locally with full F5 debugging.

Serverless Functions overview

11/19/2021 • 2 minutes to read • Edit Online

Serverless architecture evolves cloud platforms toward pure cloud-native code by abstracting code from the infrastructure that it needs to run. Azure Functions is a serverless compute option that supports *functions*, small pieces of code that do single things.

Benefits of using serverless architectures with Functions applications include:

- The Azure infrastructure automatically provides all the updated servers that applications need to keep running at scale.
- Compute resources allocate dynamically, and instantly autoscale to meet elastic demands. Serverless doesn't mean "no server," but "less server," because servers run only as needed.
- Micro-billing saves costs by charging only for the compute resources and duration the code uses to execute.
- Function *bindings* streamline integration by providing declarative access to a wide variety of Azure and third-party services.

Functions are *event-driven*. An external event like an HTTP web request, message, schedule, or change in data *triggers* the function code. A Functions application doesn't code the trigger, only the response to the trigger. With a lower barrier to entry, developers can focus on business logic, rather than writing code to handle infrastructure concerns like messaging.

Azure Functions is a managed service in Azure and Azure Stack. The open source Functions runtime works in many environments, including Kubernetes, Azure IoT Edge, on-premises, and other clouds.

Serverless and Functions require new ways of thinking and new approaches to building applications. They aren't the right solutions for every problem. For example serverless Functions scenarios, see Reference architectures.

Implementation steps

Successful implementation of serverless technologies with Azure Functions requires the following actions:

- Decide and plan

  Architects and *technical decision makers (TDMs)* perform application assessment, conduct or attend technical workshops and trainings, run proof of concept (PoC) or pilot projects, and conduct architectural designs sessions as necessary.

- Develop and deploy apps

  Developers implement serverless Functions app development patterns and practices, configure DevOps pipelines, and employ site reliability engineering (SRE) best practices.

- Manage operations

  IT professionals identify hosting configurations, future-proof scalability by automating infrastructure provisioning, and maintain availability by planning for business continuity and disaster recovery.

- Secure apps

  Security professionals handle Azure Functions security essentials, secure the hosting setup, and provide application security guidance.

Related resources

- To learn more about serverless technology, see the Azure serverless documentation.
- To learn more about Azure Functions, see the Azure Functions documentation.
- For help with choosing a compute technology, see Choose an Azure compute service for your application.

Serverless Functions reference architectures

11/19/2021 • 4 minutes to read • Edit Online

A reference architecture is a template of required components and the technical requirements to implement them. A reference architecture isn't custom-built for a customer solution, but is a high-level scenario based on extensive experience. Before designing a serverless solution, use a reference architecture to visualize an ideal technical architecture, then blend and integrate it into your environment.

Common serverless architecture patterns

Common serverless architecture patterns include:

- Serverless APIs, mobile and web backends.
- Event and stream processing, Internet of Things (IoT) data processing, big data and machine learning pipelines.
- Integration and enterprise service bus to connect line-of-business systems, publish and subscribe (Pub/Sub) to business events.
- Automation and digital transformation and process automation.
- Middleware, software-as-a-Service (SaaS) like Dynamics, and big data projects.

Web application backends Retail scenario: Pick up online orders from a queue, process them, and store the resulting data in a database

Mobile application backends Financial services scenario: Colleagues use mobile banking to reimburse each other for lunch. Whoever paid for lunch requests payment through a mobile app, triggering a notification on colleagues' phones.

IoT-connected backends Manufacturing scenario: A manufacturing company uses IoT to monitor its machines. Functions detects anomalous data and triggers a message to the service department when repair is

required.

Conversational bot processing Hospitality scenario: Customers ask for available vacation accommodations on their smartphones. A serverless bot deciphers requests and returns vacation options.

Real-time file processing Healthcare scenario: The solution securely uploads patient records as PDF files. The solution then decomposes the data, processes it using OCR detection, and adds it to a database for easy queries.

Real-time stream processing Independent software vendor (ISV) scenario: A massive cloud app collects huge amounts of telemetry data. The app processes that data in near real-time and stores it in a database for use in an analytics dashboard.

Scheduled task automation Financial services scenario: The app analyzes a customer database for duplicate entries every 15 minutes, to avoid sending out multiple communications to the same customers.

Extending SaaS applications Professional services scenario: A SaaS solution provides extensibility through webhooks, which Functions can implement to automate certain workflows.

Featured serverless reference architectures

The following featured serverless reference architectures walk through specific scenarios. See the linked articles for architectural diagrams and details.

Serverless microservices

The serverless microservices reference architecture walks you through designing, developing, and delivering the Rideshare application by Relecloud, a fictitious company. You get hands-on instructions for configuring and deploying all the architectural components, with helpful information about each component.

Serverless web application and event processing with Azure Functions

This two-part solution describes a hypothetical drone delivery system. Drones send in-flight status to the cloud, which stores these messages for later use. A web application allows users to retrieve the messages to get the latest device status.

- You can download the code for this solution from GitHub.
- The article Code walkthrough: Serverless application with Azure Functions walks you through the code and the design processes.

Event-based cloud automation

Automating workflows and repetitive tasks on the cloud can dramatically improve a DevOps team's productivity. A serverless model is best suited for event-driven automation scenarios. This event-based automation reference architecture illustrates two cloud automation scenarios: cost center tagging and throttling response.

Multicloud with Serverless Framework

The Serverless Framework architecture describes how the Microsoft Commercial Software Engineering (CSE) team partnered with a global retailer to deploy a highly-available serverless solution across both Azure and Amazon Web Services (AWS) cloud platforms, using the Serverless Framework.

More serverless Functions reference architectures

The following sections list other serverless and Azure Functions-related reference architectures and scenarios.

General

- Serverless application architectures using Event Grid
- Serverless apps using Cosmos DB
- Serverless event processing using Azure Functions
- Serverless web application on Azure
- Serverless Asynchronous Multiplayer Reference Architecture
- Instant Broadcasting on Serverless Architecture
- Building a telehealth system on Azure
- Custom Data Sovereignty & Data Gravity Requirements
- Sharing location in real time using low-cost serverless Azure services

Web and mobile backend

- An e-commerce front end
- Architect scalable e-commerce web app
- Improve scalability in an Azure web application
- Uploading and CDN-preloading static content with Azure Functions
- Cross Cloud Scaling Architecture
- Social App for Mobile and Web with Authentication

AI + Machine Learning

- Image classification for insurance claims
- Personalized Offers
- Personalized marketing solutions
- Speech transcription with Azure Cognitive Services
- Training a Model with AzureML and Azure Functions
- Customer Reviews App with Cognitive Services
- Enterprise-grade conversational bot
- AI at the Edge

- Mass ingestion and analysis of news feeds on Azure
- HIPPA and HITRUST compliant health data AI
- Intelligent Experiences On Containers (AKS, Functions, Keda)

Data and analytics

- Application integration using Event Grid
- Mass ingestion and analysis of news feeds
- Tier Applications & Data for Analytics
- Operational analysis and driving process efficiency

IoT

- Azure IoT reference (SQL DB)
- Azure IoT reference (Cosmos DB)
- IoT using Cosmos DB
- Facilities management powered by mixed reality and IoT
- Complementary Code Pattern for Azure IoT Edge Modules & Cloud Applications

Gaming

- Custom Game Server Scaling
- Non-real Time Dashboard
- In-editor Debugging Telemetry
- Multiplayer Serverless Matchmaker
- Advanced leaderboard for large scale
- Relational Leaderboard
- Content Moderation
- Text Translation
- Text to Speech
- Gaming using Cosmos DB

Automation

- Smart scaling for Azure Scale Set with Azure Functions

Plan for deploying serverless Functions

11/19/2021 • 2 minutes to read • Edit Online

To plan for moving an application to a serverless Azure Functions architecture, a technical decision maker (TDM) or architect:

- Verifies the application's characteristics and business requirements.
- Determines the application's suitability for serverless Azure Functions.
- Transforms business requirements into functional and other requirements.

Planning activities also include assessing technical team readiness, providing or attending workshops and training, and conducting architectural design reviews, proofs of concept, pilots, and technical implementations.

TDMs and architects may perform one or more of the following activities:

- **Execute an application assessment.** Evaluate the main aspects of the application to determine how complex and risky it is to rearchitect through application modernization, or rebuild a new cloud-native application. See Application assessment.
- **Attend or promote technical workshops and training.** Host a Serverless workshop or CloudHack, or enjoy many other training and learning opportunities for serverless technologies, Azure Functions, app modernization, and cloud-native apps. See Technical workshops and training.
- **Identify and execute a Proof of Concept (PoC) or pilot, or technical implementation.** Deliver a PoC, pilot, or technical implementation to provide evidence that serverless Azure Functions can solve a team's business problems. Showing teams how to modernize or build new cloud-native applications to their specifications can accelerate deployment to production. See PoC or pilot.
- **Conduct architectural design sessions.** An architectural design session (ADS) is an in-depth discussion on how a new solution will blend into the environment. ADSs validate business requirements and transform them to functional and other requirements.

Next steps

For example scenarios that use serverless architectures with Azure Functions, see Serverless reference architectures.

To move forward with serverless Azure Functions implementation, see the following resources:

- Application development and deployment
- Azure Functions app operations
- Azure Functions app security

Application assessment

11/19/2021 • 6 minutes to read • Edit Online

Cloud rationalization is the process of evaluating applications to determine the best way to migrate or modernize them for the cloud.

Rationalization methods include:

- **Rehost**. Also known as a *lift and shift* migration, rehost moves a current application to the cloud with minimal change.
- **Refactor**. Slightly refactoring an application to fit *platform-as-a-service* (PaaS)-based options can reduce operational costs.
- **Rearchitect**. Rearchitect aging applications that aren't compatible with cloud components, or cloud-compatible applications that would realize cost and operational efficiencies by rearchitecting into a cloud-native solution.
- **Rebuild**. If the changes or costs to carry an application forward are too great, consider creating a new cloud-native code base. Rebuild is especially appropriate for applications that previously met business needs, but are now unsupported or misaligned with current business processes.

Before you decide on an appropriate strategy, analyze the current application to determine the risk and complexity of each method. Consider application lifecycle, technology, infrastructure, performance, and operations and monitoring. For multitier architectures, evaluate the presentation tier, service tier, integrations tier, and data tier.

The following checklists evaluate an application to determine the complexity and risk of rearchitecting or rebuilding.

Complexity and risk

Each of the following factors adds to complexity, risk, or both.

Architecture

Define the high-level architecture, such as web application, web services, data storage, or caching.

FACTOR	COMPLEXITY	RISK
Application components don't translate directly to Azure.	✔	✔
The application needs code changes to run in Azure.	✔	✔
The application needs major, complex code changes to run in Azure.	✔	✔

Business drivers

Older applications might require extensive changes to get to the cloud.

FACTOR	COMPLEXITY	RISK
This application has been around for more than three years.	✔	
This application is business critical.		✔
There are technology blockers for migration.	✔	
There are business blockers for migration.	✔	
This application has compliance requirements.		✔
The application is subject to country-specific data requirements.		✔
The application is publicly accessible.	✔	✔

Technology

FACTOR	COMPLEXITY	RISK
This is not a web-based application, and isn't hosted on a web server.	✔	
The app isn't hosted in Windows IIS	✔	
The app isn't hosted on Linux	✔	
The application is hosted in a web farm, and requires multiple servers to host the web components.	✔	✔
The application requires third-party software to be installed on the servers.	✔	✔
The application is hosted in a single datacenter, and operations are performed in a single location.	✔	
The application accesses the server's registry.	✔	
The application sends emails, and needs access to an SMTP server.	✔	
This isn't a .NET application.	✔	
The application uses SQL Server as its data store.	✔	

FACTOR	COMPLEXITY	RISK
The application stores data on local disks, and needs access to the disks to operate properly.	✔	
The application uses Windows Services to process asynchronous operations, or needs external services to process data or operations.	✔	

Deployment

When assessing deployment requirements, consider:

- Number of daily users
- Average number of concurrent users
- Expected traffic
- Bandwidth in Gbps
- Requests per second
- Amount of memory needed

You can reduce deployment risk by storing code under source control in a version control system such as Git, Azure DevOps Server, or SVN.

FACTOR	COMPLEXITY	RISK
Leveraging existing code and data is a #1 priority.	✔	✔
The application code isn't under source control.		✔
There's no automated build process like Azure DevOps Server or Jenkins.		✔
There's no automated release process to deploy the application.	✔	✔
The application has a Service Level Agreement (SLA) that dictates the amount of expected downtime.		✔
The application experiences peak or variable usage times or loads.		✔
The web application saves its session state in process, rather than an external data store.	✔	

Operations

FACTOR	COMPLEXITY	RISK
The application doesn't have a well-established instrumentation strategy or standard instrumentation framework.		✔
The application doesn't use monitoring tools, and the operations team doesn't monitor the app's performance.		✔
The application has measured SLA in place, and the operations team monitors the application's performance.		✔
The application writes to a log store, event log, log file, log database, or Application Insights.	✔	
The application doesn't write to a log store, event log, log file, log database, or Application Insights.		✔
The application isn't part of the organization's disaster recovery plan.		✔

Security

FACTOR	COMPLEXITY	RISK
The application uses Active Directory to authenticate users.	✔	✔
The organization hasn't yet configured Azure Active Directory (Azure AD), or hasn't configured Azure AD Connect to synchronize on-premises AD with Azure AD.	✔	
The application requires access to on-premises resources, which will require VPN connectivity from Azure.	✔	
The organization hasn't yet configured a VPN connection between Azure and their on-premises environment.	✔	✔
The application requires a SSL certificate to run.	✔	✔

Results

Count your application's **Complexity** and **Risk** checkmarks.

- The expected level of complexity to migrate or modernize the application to Azure is: **Total Complexity/25**.
- The expected risk involved is: **Total Risk/19**.

For both complexity and risk, a score of <0.3 = low, <0.7 = medium, >0.7 = high.

Refactor, rearchitect, or rebuild

To rationalize whether to rehost, refactor, rearchitect, or rebuild your application, consider the following points. Many of these factors also contribute to complexity and risk.

Determine whether the application components can translate directly to Azure. If so, you don't need code changes to move the application to Azure, and could use rehost or refactor strategies. If not, you need to rewrite code, so you need to rearchitect or rebuild.

If the app does need code changes, determine the complexity and extent of the needed changes. Minor changes might allow for rearchitecting, while major changes may require rebuilding.

Rehost or refactor

- If leveraging existing code and data is a top priority, consider a refactor strategy rather than rearchitecting or rebuilding.
- If you have pressing timelines like datacenter shutdown or contract expiration, end-of-life licensing, or mergers or acquisitions, the fastest way to get the application to Azure might be to rehost, followed by refactoring to take advantage of cloud capabilities.

Rearchitect or rebuild

- If there are applications serving similar needs in your portfolio, this might be an opportunity to rearchitect or rebuild the entire solution.
- If you want to implement multi-tier or microservices architecture for a monolithic app, you must rearchitect or rebuild the app. If you don't mind retaining the monolithic structure, you might be able to rehost or refactor.
- Rearchitect or rebuild the app to take advantage of cloud capabilities if you plan to update the app more often than yearly, if the app has peak or variable usage times, or if you expect the app to handle high traffic.

To decide between rearchitecting or rebuilding, assess the following factors. The largest scoring result indicates your best strategy.

FACTOR	REARCHITECT	REBUILD
There are other applications serving similar needs in your portfolio.	✔	✔
The application needs minor code changes to run in Azure.	✔	
The application needs major, complex code changes to run in Azure.		✔
It's important to leverage existing code.	✔	
You want to move a monolithic application to multi-tier architecture.	✔	

FACTOR	REARCHITECT	REBUILD
You want to move a monolithic application to a microservices architecture.	✔	✔
You expect this app to add breakthrough capabilities like AI, IoT, or bots.		✔
Among functionality, cost, infrastructure, and processes, functionality is the least efficient aspect of this application.		✔
The application requires third-party software installed on the servers.	✔	
The application accesses the server's registry.	✔	
The application sends emails and needs access to an SMTP server.	✔	
The application uses SQL Server as its data store.	✔	
The application stores data on local disks, and needs access to the disks to run properly.	✔	
The application uses Windows services to process asynchronous operations, or needs external services to process data or operations.	✔	
A web application saves its session state in process, rather than to an external data store.	✔	
The app has peak and variable usage times and loads.	✔	✔
You expect the application to handle high traffic.	✔	✔

Technical workshops and training

11/19/2021 • 3 minutes to read • Edit Online

The workshops, classes, and learning materials in this article provide technical training for serverless architectures with Azure Functions. These resources help you and your team or customers understand and implement application modernization and cloud-native apps.

Technical workshops

The Microsoft Cloud Workshop (MCW) program provides workshops you can host to foster cloud learning and adoption. Each workshop includes presentation decks, trainer and student guides, and hands-on lab guides. Contribute your own content and feedback to add to a robust database of training guides for deploying advanced Azure workloads on the Microsoft Cloud Platform.

Workshops related to application development workloads include:

- Serverless APIs in Azure. Set of entry-level exercises, which cover the basics of building and managing serverless APIs in Microsoft Azure - with Azure Functions, Azure API Management, and Azure Application Insights.
- Serverless architecture. Implement a series of Azure Functions that independently scale and break down business logic to discrete components, allowing customers to pay only for the services they use.
- App modernization. Design a modernization plan to move services from on-premises to the cloud by leveraging cloud, web, and mobile services, secured by Azure Active Directory.
- Modern cloud apps. Deploy, configure, and implement an end-to-end secure and Payment Card Industry (PCI) compliant solution for e-commerce, based on Azure App Services, Azure Active Directory, and Azure DevOps.
- Cloud-native applications. Using DevOps best practices, build a proof of concept (PoC) to transform a platform-as-a-service (PaaS) application to a container-based application with multi-tenant web app hosting.
- Continuous delivery in Azure DevOps. Set up and configure continuous delivery (CD) in Azure to reduce manual errors, using Azure Resource Manager templates, Azure DevOps, and Git repositories for source control.

Instructor-led training

Course AZ-204: Developing solutions for Microsoft Azure teaches developers how to create end-to-end solutions in Microsoft Azure. Students learn how to implement Azure compute solutions, create Azure Functions, implement and manage web apps, develop solutions utilizing Azure Storage, implement authentication and authorization, and secure their solutions by using Azure Key Vault and managed identities. Students also learn to connect to and consume Azure and third-party services, and include event- and message-based models in their solutions. The course also covers monitoring, troubleshooting, and optimizing Azure solutions.

Serverless OpenHack

The Serverless OpenHack simulates a real-world scenario where a company wants to utilize serverless services to build and release an API to integrate into their distributor's application. This OpenHack lets attendees quickly build and deploy Azure serverless solutions with cutting-edge compute services like Azure Functions, Logic Apps, Event Grid, Service Bus, Event Hubs, and Cosmos DB. The OpenHack also covers related technologies like API Management, Azure DevOps or GitHub, Application Insights, Dynamics 365/Microsoft 365, and Cognitive APIs.

OpenHack attendees build serverless functions, web APIs, and a CI/CD pipeline to support them. They implement further serverless technologies to integrate line of business (LOB) app workflows, process user and data telemetry, and create key progress indicator (KPI)-aligned reports. By the end of the OpenHack, attendees have built out a full serverless technical solution that can create workflows between systems and handle events, files, and data ingestion.

Microsoft customer projects inspired these OpenHack challenges:

- Configure the developer environment.
- Create your first serverless function and workflow.
- Build APIs to support business needs.
- Deploy a management layer for APIs and monitoring usage.
- Build a LOB workflow process.
- Process large amounts of unstructured file data.
- Process large amounts of incoming event data.
- Implement a publisher/subscriber messaging pattern and virtual network integration.
- Conduct sentiment analysis.
- Perform data aggregation, analysis, and reporting.

To attend an OpenHack, register at https://openhack.microsoft.com. For enterprises with many engineers, Microsoft can request and organize a dedicated Serverless OpenHack.

Microsoft Learn

Microsoft Learn is a free, online training platform that provides interactive learning for Microsoft products. The goal is to improve proficiency with fun, guided, hands-on content that's specific to your role and goals. Learning paths are collections of modules that are organized around specific roles like developer, architect, or system admin, or technologies like Azure Web Apps, Azure Functions, or Azure SQL DB. Learning paths provide understanding of different aspects of the technology or role.

Learning paths about serverless apps and Azure Functions include:

- Create serverless applications. Learn how to leverage functions to execute server-side logic and build serverless architectures.
- Architect message brokering and serverless applications in Azure. Learn how to create reliable messaging for your applications, and how to take advantage of serverless application services in Azure.
- Search all Functions-related learning paths.

Hands-on labs and how-to guides

- Build a serverless web app, from the Build 2018 conference
- Build a Serverless IoT Solution with Python Azure Functions and SignalR

Next steps

- Execute an application assessment
- Identify and execute a PoC or Pilot project

Proof of concept or pilot

11/19/2021 • 4 minutes to read • Edit Online

When driving a technical and security decision for your company or customer, a *Proof of Concept (PoC)* or *pilot* is an opportunity to deliver evidence that the proposed solution solves the business problems. The PoC or pilot increases the likelihood of a successful adoption.

A PoC:

- Demonstrates that a business model or idea is feasible and will work to solve the business problem
- Usually involves one to three features or capabilities
- Can be in one or multiple technologies
- Is usually geared toward a particular scenario, and proves what the customer needs to know to make the technical or security decision
- Is used only as a demonstration and won't go into production
- Is IT-driven and enablement-driven

A pilot:

- Is a test run or trial of a proposed action or product
- Lasts longer than a PoC, often weeks or months
- Has a higher return on investment (ROI) than a PoC
- Builds in a pre-production or trial environment, with the intent that it will then go into production
- Is adoption-driven and consumption-driven

PoC and pilot best practices

Be aware of compliance issues when working in a customer's environment, and make sure your actions are always legal and compliant.

- Touching or altering the customer's environment usually requires a contract, and may involve a partner or Microsoft services. Without a contract, your company may be liable for issues or damages.
- Governance may require Legal department approval. Your company may not be able to give intellectual property (IP) away for free. You may need a legal contract or contracts to specify whether your company or the customer pays for the IP.
- Get disclosure guidance when dealing with non-disclosure agreements (NDAs), product roadmaps, NDA features, or anything not released to the general public.
- In a pilot, don't use a trial Microsoft Developers Network (MSDN) environment, or any environment that you own.
- Use properly-licensed software, and ask the opportunity owner to make sure to handle software licensing correctly.

The customer, partner, or your company may pay for the PoC or pilot. Depending on the size of the contract, the ROI, and the cost of sale, one group may cover it all, or a combination of all three parties may cover the cost. Ensure that your company or customer has some investment in the PoC or pilot. If they don't, this can be a red flag signaling that your company or customer doesn't yet see value in the solution.

PoC and pilot process

The Technical Decision Maker (TDM) is responsible for driving an adoption decision. The TDM is responsible for

ensuring that the right partners and resources are involved in a PoC or pilot. As a TDM, make sure you're aware of the partners in your product and service area or region. Be aware of their key service offerings around your product service area.

Planning

Consider the following health questions:

- Do you have a good technical plan, including key decision makers and Microsoft potential?
- Can you deliver the needed assurance without a PoC?
- Should you switch to a pilot?
- What are the detailed scope and decision criteria your team or customer agreed to?
- If you meet the criteria, will your company or customer buy or deploy the solution?

Do the following tasks:

- Analyze risk.
- Evaluate the setting.
- Do the preparation.
- Consider workloads and human resources.
- Present PoC or pilot health status.
- Fulfill technical prerequisites.
- Define the go/no go decision.
- Create a final project plan specification.

Execution

For the execution phase:

- Determine who kicks off the presentation.
- Schedule the meeting in the morning, if possible.
- Prepare the demos and slides.
- Conduct a dry run to refine the presentation.
- Get feedback.
- Involve your company or customer team.
- Complete the win/lose statement.

Debriefing

During the debriefing phase, consider:

- Whether criteria were met or not met
- Stakeholder investment
- Initiating deployment
- Finding a partner and training
- Lessons learned
- Corrections or extensions of PoC or pilot guidance
- Archiving of valuable deliverables

Change management

Change management uses tested methods and techniques to avoid errors and minimize impact when administering change.

Ideally, a pilot includes a cross-section of users, to address any potential issues or problems that arise. Users may be comfortable and familiar with their old technology, and have difficulty moving into new technical

solutions. Change management keeps this in mind, and helps the user understand the reasons behind the change and the impact the change will make.

This understanding is part of a pilot, and addresses everyone who has a stake in the project. A pilot is better than a PoC, because the customer is more involved, so they're more likely to implement the change.

The pilot includes a detailed follow up through surveys or focus groups. The feedback can prove and improve the change.

Next steps

- Execute an application assessment
- Promote a technical workshop or training
- Code a technical implementation with the team or customer

Related resources

Prosci® change management training

Application development and deployment

11/19/2021 • 4 minutes to read • Edit Online

To develop and deploy serverless applications with Azure Functions, examine patterns and practices, configure DevOps pipelines, and implement site reliability engineering (SRE) best practices.

For detailed information about serverless architectures and Azure Functions, see:

- Serverless apps: Architecture, patterns, and Azure implementation
- Azure Serverless Computing Cookbook
- Example serverless reference architectures

Planning

To plan app development and deployment:

1. Prepare development environment and set up workflow.
2. Structure projects to support Azure Functions app development.
3. Identify app triggers, bindings, and configuration requirements.

Understand event-driven architecture

A different event triggers every function in a serverless Functions project. For more information about event-driven architectures, see:

- Event-driven architecture style.
- Event-driven design patterns to enhance existing applications using Azure Functions

Prepare development environment

Set up your development workflow and environment with the tools to create Functions. For details about development tools and Functions code project structure, see:

- Code and test Azure Functions locally
- Develop Azure Functions by using Visual Studio Code
- Develop Azure Functions using Visual Studio
- Work with Azure Functions Core Tools
- Folder structure

Development

Decide on the development language to use. Azure Functions supports C#, F#, PowerShell, JavaScript, TypeScript, Java, and Python. All of a project's Functions must be in the same language. For more information, see Supported languages in Azure Functions.

Define triggers and bindings

A trigger invokes a Function, and every Function must have exactly one trigger. Binding to a Function declaratively connects another resource to the Function. For more information about Functions triggers and bindings, see:

- Azure Functions triggers and bindings concepts
- Execute an Azure Function with triggers
- Chain Azure Functions together using input and output bindings

Create the Functions application

Functions follow the single responsibility principle: do only one thing. For more information about Functions development, see:

- Azure Functions developers guide
- Create serverless applications
- Strategies for testing your code in Azure Functions
- Functions best practices

Use Durable Functions for stateful workflows

Durable Functions in Azure Functions let you define stateful workflows in a serverless environment by writing *orchestrator functions*, and stateful entities by writing *entity functions*. Durable Functions manage state, checkpoints, and restarts, allowing you to focus on business logic. For more information, see What are Durable Functions.

Understand and address cold starts

If the number of serverless host instances scales down to zero, the next request has the added latency of restarting the Function app, called a *cold start*. To minimize the performance impact of cold starts, reduce dependencies that the Functions app needs to load on startup, and use as few large, synchronous calls and operations as possible. For more information about autoscaling and cold starts, see Serverless Functions operations.

Manage application secrets

For security, don't store credentials in application code. To use Azure Key Vault with Azure Functions to store and retrieve keys and credentials, see Use Key Vault references for App Service and Azure Functions.

For more information about serverless Functions application security, see Serverless Functions security.

Deployment

To prepare serverless Functions application for production, make sure you can:

- Fulfill application resource requirements.
- Monitor all aspects of the application.
- Diagnose and troubleshoot application issues.
- Deploy new application versions without affecting production systems.

Define deployment technology

Decide on deployment technology, and organize scheduled releases. For more information about how Functions app deployment enables reliable, zero-downtime upgrades, see Deployment technologies in Azure Functions.

Avoid using too many resource connections

Functions in a Functions app share resources, including connections to HTTPS, databases, and services such as Azure Storage. When many Functions are running concurrently, it's possible to run out of available connections. For more information, see Manage connections in Azure Functions.

Configure logging, alerting, and application monitoring

Application Insights in Azure Monitor collects log, performance, and error data. Application Insights automatically detects performance anomalies, and includes powerful analytics tools to help diagnose issues and understand function usage.

For more information about application monitoring and logging, see:

- Monitor Azure Functions
- Monitoring Azure Functions with Azure Monitor Logs

- Application Insights for Azure Functions supported features

Diagnose and troubleshoot issues

Learn how to effectively use diagnostics for troubleshooting in proactive and problem-first scenarios. For more information, see:

- Keep your Azure App Service and Azure Functions apps healthy and happy
- Troubleshoot error: "Azure Functions Runtime is unreachable"

Deploy applications using an automated pipeline and DevOps

Full automation of all steps from code commit to production deployment lets teams focus on building code, and removes the overhead and potential human error of manual steps. Deploying new code is quicker and less risky, helping teams become more agile, more productive, and more confident about their code.

For more information about DevOps and continuous deployment (CD), see:

- Continuous deployment for Azure Functions
- Continuous delivery by using Azure DevOps
- Continuous delivery by using GitHub Action

Optimization

Once the application is in production, prepare for scaling and implement site reliability engineering (SRE).

Ensure optimal scalability

For information about factors that impact Functions app scalability, see:

- Scalability best practices
- Performance and scale in Durable Functions

Implement SRE practices

Site Reliability Engineering (SRE) is a proven approach to maintaining crucial system and application reliability, while iterating at the speed the marketplace demands. For more information, see:

- Introduction to Site Reliability Engineering (SRE)
- DevOps at Microsoft: Game streaming SRE

Next steps

For hands-on serverless Functions app development and deployment walkthroughs, see:

- Serverless Functions code walkthrough
- CI/CD for a serverless frontend

For an engineering playbook to help teams and customers successfully implement serverless Functions projects, see the Code-With Customer/Partner Engineering Playbook.

Code walkthrough: Serverless application with Functions

11/19/2021 • 18 minutes to read • Edit Online

Serverless models abstract code from the underlying compute infrastructure, allowing developers to focus on business logic without extensive setup. Serverless code reduces costs, because you pay only for the code execution resources and duration.

The serverless event-driven model fits situations where a certain event triggers a defined action. For example, receiving an incoming device message triggers storage for later use, or a database update triggers some further processing.

To help you explore Azure serverless technologies in Azure, Microsoft developed and tested a serverless application that uses Azure Functions. This article walks through the code for the serverless Functions solution, and describes design decisions, implementation details, and some of the "gotchas" you might encounter.

Explore the solution

The two-part solution describes a hypothetical drone delivery system. Drones send in-flight status to the cloud, which stores these messages for later use. A web app lets users retrieve the messages to get the latest status of the devices.

You can download the code for this solution from GitHub.

This walkthrough assumes basic familiarity with the following technologies:

- Azure Functions
- Azure Event Hubs
- .NET Core

You don't need to be an expert in Functions or Event Hubs, but you should understand their features at a high level. Here are some good resources to get started:

- An introduction to Azure Functions
- Features and terminology in Azure Event Hubs

Understand the scenario

Fabrikam manages a fleet of drones for a drone delivery service. The application consists of two main functional areas:

- **Event ingestion**. During flight, drones send status messages to a cloud endpoint. The application ingests and processes these messages, and writes the results to a back-end database (Cosmos DB). The devices send messages in protocol buffer (protobuf) format. Protobuf is an efficient, self-describing serialization format.

These messages contain partial updates. At a fixed interval, each drone sends a "key frame" message that contains all of the status fields. Between key frames, the status messages only include fields that changed since the last message. This behavior is typical of many IoT devices that need to conserve bandwidth and power.

- **Web app**. A web application allows users to look up a device and query the device's last-known status. Users must sign into the application and authenticate with Azure Active Directory (Azure AD). The application only allows requests from users who are authorized to access the app.

Here's a screenshot of the web app, showing the result of a query:

Design the application

Fabrikam has decided to use Azure Functions to implement the application business logic. Azure Functions is an example of "Functions as a Service" (FaaS). In this computing model, a *function* is a piece of code that is deployed to the cloud and runs in a hosting environment. This hosting environment completely abstracts the servers that run the code.

Why choose a serverless approach?

A serverless architecture with Functions is an example of an event-driven architecture. The function code is a triggered by some event that's external to the function — in this case, either a message from a drone, or an HTTP request from a client application. With a function app, you don't need to write any code for the trigger. You only write the code that runs in response to the trigger. That means you can focus on your business logic, rather than writing a lot of code to handle infrastructure concerns like messaging.

There are also some operational advantages to using a serverless architecture:

- There is no need to manage servers.
- Compute resources are allocated dynamically as needed.
- You are charged only for the compute resources used to execute your code.
- The compute resources scale on demand based on traffic.

Architecture

The following diagram shows the high-level architecture of the application:

In one data flow, arrows show messages going from Devices to Event Hubs and triggering the function app. From the app, one arrow shows dead-letter messages going to a storage queue, and another arrow shows writing to Azure Cosmos DB. In another data flow, arrows show the client web app getting static files from Blob storage static web hosting, through a CDN. Another arrow shows the client HTTP request going through API Management. From API Management, one arrow shows the function app triggering and reading data from Azure Cosmos DB. Another arrow shows authentication through Azure AD. A User also signs in to Azure AD.

Event ingestion:

1. Drone messages are ingested by Azure Event Hubs.
2. Event Hubs produces a stream of events that contain the message data.
3. These events trigger an Azure Functions app to process them.
4. The results are stored in Cosmos DB.

Web app:

1. Static files are served by CDN from Blob storage.
2. A user signs into the web app using Azure AD.
3. Azure API Management acts as a gateway that exposes a REST API endpoint.
4. HTTP requests from the client trigger an Azure Functions app that reads from Cosmos DB and returns the result.

This application is based on two reference architectures, corresponding to the two functional blocks described above:

- Serverless event processing using Azure Functions
- Serverless web application on Azure

You can read those articles to learn more about the high-level architecture, the Azure services that are used in the solution, and considerations for scalability, security, and reliability.

Drone telemetry function

Let's start by looking at the function that processes drone messages from Event Hubs. The function is defined in a class named `RawTelemetryFunction`:

```
namespace DroneTelemetryFunctionApp
{
    public class RawTelemetryFunction
    {
        private readonly ITelemetryProcessor telemetryProcessor;
        private readonly IStateChangeProcessor stateChangeProcessor;
        private readonly TelemetryClient telemetryClient;

        public RawTelemetryFunction(ITelemetryProcessor telemetryProcessor, IStateChangeProcessor
stateChangeProcessor, TelemetryClient telemetryClient)
        {
            this.telemetryProcessor = telemetryProcessor;
            this.stateChangeProcessor = stateChangeProcessor;
            this.telemetryClient = telemetryClient;
        }
    }
    ...
}
```

This class has several dependencies, which are injected into the constructor using dependency injection:

- The `ITelemetryProcessor` and `IStateChangeProcessor` interfaces define two helper objects. As we'll see, these objects do most of the work.
- The TelemetryClient is part of the Application Insights SDK. It is used to send custom application metrics to Application Insights.

Later, we'll look at how to configure the dependency injection. For now, just assume these dependencies exist.

Configure the Event Hubs trigger

The logic in the function is implemented as an asynchronous method named `RunAsync`. Here is the method signature:

```
[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]
public async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection", ConsumerGroup
="%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage> deadLetterMessages,
    ILogger logger)
{
    // implementation goes here
}
```

The method takes the following parameters:

- `messages` is an array of event hub messages.
- `deadLetterMessages` is an Azure Storage Queue, used for storing dead letter messages.
- `logging` provides a logging interface, for writing application logs. These logs are sent to Azure Monitor.

The `EventHubTrigger` attribute on the `messages` parameter configures the trigger. The properties of the attribute specify an event hub name, a connection string, and a consumer group. (A *consumer group* is an isolated view of the Event Hubs event stream. This abstraction allows for multiple consumers of the same event hub.)

Notice the percent signs (%) in some of the attribute properties. These indicate that the property specifies the name of an app setting, and the actual value is taken from that app setting at run time. Otherwise, without percent signs, the property gives the literal value.

The `Connection` property is an exception. This property always specifies an app setting name, never a literal value, so the percent sign is not needed. The reason for this distinction is that a connection string is secret and should never be checked into source code.

While the other two properties (event hub name and consumer group) are not sensitive data like a connection string, it's still better to put them into app settings, rather than hard coding. That way, they can be updated without recompiling the app.

For more information about configuring this trigger, see Azure Event Hubs bindings for Azure Functions.

Message processing logic

Here's the implementation of the `RawTelemetryFunction.RunAsync` method that processes a batch of messages:

```
[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]
public async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection", ConsumerGroup
="%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage> deadLetterMessages,
    ILogger logger)
{
    telemetryClient.GetMetric("EventHubMessageBatchSize").TrackValue(messages.Length);

    foreach (var message in messages)
    {
        DeviceState deviceState = null;

        try
        {
            deviceState = telemetryProcessor.Deserialize(message.Body.Array, logger);

            try
            {
                await stateChangeProcessor.UpdateState(deviceState, logger);
            }
            catch (Exception ex)
            {
                logger.LogError(ex, "Error updating status document", deviceState);
                await deadLetterMessages.AddAsync(new DeadLetterMessage { Exception = ex, EventData =
message, DeviceState = deviceState });
            }
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Error deserializing message", message.SystemProperties.PartitionKey,
message.SystemProperties.SequenceNumber);
            await deadLetterMessages.AddAsync(new DeadLetterMessage { Exception = ex, EventData = message
});
        }
    }
}
```

When the function is invoked, the `messages` parameter contains an array of messages from the event hub. Processing messages in batches will generally yield better performance than reading one message at a time. However, you have to make sure the function is resilient and handles failures and exceptions gracefully. Otherwise, if the function throws an unhandled exception in the middle of a batch, you might lose the remaining messages. This consideration is discussed in more detail in the section Error handling.

But if you ignore the exception handling, the processing logic for each message is simple:

1. Call `ITelemetryProcessor.Deserialize` to deserialize the message that contains a device state change.

2. Call `IStateChangeProcessor.UpdateState` to process the state change.

Let's look at these two methods in more detail, starting with the `Deserialize` method.

Deserialize method

The `TelemetryProcess.Deserialize` method takes a byte array that contains the message payload. It deserializes this payload and returns a `DeviceState` object, which represents the state of a drone. The state may represent a partial update, containing just the delta from the last-known state. Therefore, the method needs to handle `null` fields in the deserialized payload.

```
public class TelemetryProcessor : ITelemetryProcessor
{
    private readonly ITelemetrySerializer<DroneState> serializer;

    public TelemetryProcessor(ITelemetrySerializer<DroneState> serializer)
    {
        this.serializer = serializer;
    }

    public DeviceState Deserialize(byte[] payload, ILogger log)
    {
        DroneState restored = serializer.Deserialize(payload);

        log.LogInformation("Deserialize message for device ID {DeviceId}", restored.DeviceId);

        var deviceState = new DeviceState();
        deviceState.DeviceId = restored.DeviceId;

        if (restored.Battery != null)
        {
            deviceState.Battery = restored.Battery;
        }
        if (restored.FlightMode != null)
        {
            deviceState.FlightMode = (int)restored.FlightMode;
        }
        if (restored.Position != null)
        {
            deviceState.Latitude = restored.Position.Value.Latitude;
            deviceState.Longitude = restored.Position.Value.Longitude;
            deviceState.Altitude = restored.Position.Value.Altitude;
        }
        if (restored.Health != null)
        {
            deviceState.AccelerometerOK = restored.Health.Value.AccelerometerOK;
            deviceState.GyrometerOK = restored.Health.Value.GyrometerOK;
            deviceState.MagnetometerOK = restored.Health.Value.MagnetometerOK;
        }
        return deviceState;
    }
}
```

This method uses another helper interface, `ITelemetrySerializer<T>`, to deserialize the raw message. The results are then transformed into a POCO model that is easier to work with. This design helps to isolate the processing logic from the serialization implementation details. The `ITelemetrySerializer<T>` interface is defined in a shared library, which is also used by the device simulator to generate simulated device events and send them to Event Hubs.

```
using System;

namespace Serverless.Serialization
{
    public interface ITelemetrySerializer<T>
    {
        T Deserialize(byte[] message);

        ArraySegment<byte> Serialize(T message);
    }
}
```

UpdateState method

The `StateChangeProcessor.UpdateState` method applies the state changes. The last-known state for each drone is stored as a JSON document in Cosmos DB. Because the drones send partial updates, the application can't simply overwrite the document when it gets an update. Instead, it needs to fetch the previous state, merge the fields, and then perform an upsert operation.

```
public class StateChangeProcessor : IStateChangeProcessor
{
    private IDocumentClient client;
    private readonly string cosmosDBDatabase;
    private readonly string cosmosDBCollection;

    public StateChangeProcessor(IDocumentClient client, IOptions<StateChangeProcessorOptions> options)
    {
        this.client = client;
        this.cosmosDBDatabase = options.Value.COSMOSDB_DATABASE_NAME;
        this.cosmosDBCollection = options.Value.COSMOSDB_DATABASE_COL;
    }

    public async Task<ResourceResponse<Document>> UpdateState(DeviceState source, ILogger log)
    {
        log.LogInformation("Processing change message for device ID {DeviceId}", source.DeviceId);

        DeviceState target = null;

        try
        {
            var response = await client.ReadDocumentAsync(UriFactory.CreateDocumentUri(cosmosDBDatabase,
cosmosDBCollection, source.DeviceId),
                                                                  new RequestOptions { PartitionKey = new
PartitionKey(source.DeviceId) });

            target = (DeviceState)(dynamic)response.Resource;

            // Merge properties
            target.Battery = source.Battery ?? target.Battery;
            target.FlightMode = source.FlightMode ?? target.FlightMode;
            target.Latitude = source.Latitude ?? target.Latitude;
            target.Longitude = source.Longitude ?? target.Longitude;
            target.Altitude = source.Altitude ?? target.Altitude;
            target.AccelerometerOK = source.AccelerometerOK ?? target.AccelerometerOK;
            target.GyrometerOK = source.GyrometerOK ?? target.GyrometerOK;
            target.MagnetometerOK = source.MagnetometerOK ?? target.MagnetometerOK;
        }
        catch (DocumentClientException ex)
        {
            if (ex.StatusCode == System.Net.HttpStatusCode.NotFound)
            {
                target = source;
            }
        }

        var collectionLink = UriFactory.CreateDocumentCollectionUri(cosmosDBDatabase, cosmosDBCollection);
        return await client.UpsertDocumentAsync(collectionLink, target);
    }
}
```

This code uses the `IDocumentClient` interface to fetch a document from Cosmos DB. If the document exists, the new state values are merged into the existing document. Otherwise, a new document is created. Both cases are handled by the `UpsertDocumentAsync` method.

This code is optimized for the case where the document already exists and can be merged. On the first telemetry message from a given drone, the `ReadDocumentAsync` method will throw an exception, because there is no document for that drone. After the first message, the document will be available.

Notice that this class uses dependency injection to inject the `IDocumentClient` for Cosmos DB and an `IOptions<T>` with configuration settings. We'll see how to set up the dependency injection later.

> **NOTE**
> Azure Functions supports an output binding for Cosmos DB. This binding lets the function app write documents in Cosmos DB without any code. However, the output binding won't work for this particular scenario, because of the custom upsert logic that's needed.

Error handling

As mentioned earlier, the `RawTelemetryFunction` function app processes a batch of messages in a loop. That means the function needs to handle any exceptions gracefully and continue processing the rest of the batch. Otherwise, messages might get dropped.

If an exception is encountered when processing a message, the function puts the message onto a dead-letter queue:

```
catch (Exception ex)
{
    logger.LogError(ex, "Error deserializing message", message.SystemProperties.PartitionKey,
message.SystemProperties.SequenceNumber);
    await deadLetterMessages.AddAsync(new DeadLetterMessage { Exception = ex, EventData = message });
}
```

The dead-letter queue is defined using an output binding to a storage queue:

```
[FunctionName("RawTelemetryFunction")]
[StorageAccount("DeadLetterStorage")]  // App setting that holds the connection string
public async Task RunAsync(
    [EventHubTrigger("%EventHubName%", Connection = "EventHubConnection", ConsumerGroup
="%EventHubConsumerGroup%")]EventData[] messages,
    [Queue("deadletterqueue")] IAsyncCollector<DeadLetterMessage> deadLetterMessages,  // output binding
    ILogger logger)
```

Here the `Queue` attribute specifies the output binding, and the `StorageAccount` attribute specifies the name of an app setting that holds the connection string for the storage account.

Deployment tip: In the Resource Manager template that creates the storage account, you can automatically populate an app setting with the connection string. The trick is to use the listKeys function.

Here is the section of the template that creates the storage account for the queue:

```
    {
        "name": "[variables('droneTelemetryDeadLetterStorageQueueAccountName')]",
        "type": "Microsoft.Storage/storageAccounts",
        "location": "[resourceGroup().location]",
        "apiVersion": "2017-10-01",
        "sku": {
            "name": "[parameters('storageAccountType')]"
        },
```

Here is the section of the template that creates the function app.

```
    {
        "apiVersion": "2015-08-01",
        "type": "Microsoft.Web/sites",
        "name": "[variables('droneTelemetryFunctionAppName')]",
        "location": "[resourceGroup().location]",
        "tags": {
            "displayName": "Drone Telemetry Function App"
        },
        "kind": "functionapp",
        "dependsOn": [
            "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
            ...
        ],
        "properties": {
            "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables('hostingPlanName'))]",
            "siteConfig": {
                "appSettings": [
                    {
                        "name": "DeadLetterStorage",
                        "value": "[concat('DefaultEndpointsProtocol=https;AccountName=',
variables('droneTelemetryDeadLetterStorageQueueAccountName'), ';AccountKey=',
listKeys(variables('droneTelemetryDeadLetterStorageQueueAccountId'),'2015-05-01-preview').key1)]"
                    },
                    ...
```

This defines an app setting named `DeadLetterStorage` whose value is populated using the `listKeys` function. It's important to make the function app resource depend on the storage account resource (see the `dependsOn` element). This guarantees that the storage account is created first and the connection string is available.

Setting up dependency injection

The following code sets up dependency injection for the `RawTelemetryFunction` function:

```
[assembly: FunctionsStartup(typeof(DroneTelemetryFunctionApp.Startup))]

namespace DroneTelemetryFunctionApp
{
    public class Startup : FunctionsStartup
    {
        public override void Configure(IFunctionsHostBuilder builder)
        {
            builder.Services.AddOptions<StateChangeProcessorOptions>()
                .Configure<IConfiguration>((configSection, configuration) =>
                {
                    configuration.Bind(configSection);
                });

            builder.Services.AddTransient<ITelemetrySerializer<DroneState>, TelemetrySerializer<DroneState>>
();
            builder.Services.AddTransient<ITelemetryProcessor, TelemetryProcessor>();
            builder.Services.AddTransient<IStateChangeProcessor, StateChangeProcessor>();

            builder.Services.AddSingleton<IDocumentClient>(ctx => {
                var config = ctx.GetService<IConfiguration>();
                var cosmosDBEndpoint = config.GetValue<string>("CosmosDBEndpoint");
                var cosmosDBKey = config.GetValue<string>("CosmosDBKey");
                return new DocumentClient(new Uri(cosmosDBEndpoint), cosmosDBKey);
            });
        }
    }
}
```

Azure Functions written for .NET can use the ASP.NET Core dependency injection framework. The basic idea is that you declare a startup method for your assembly. The method takes an `IFunctionsHostBuilder` interface, which is used to declare the dependencies for DI. You do this by calling `Add*` method on the `Services` object. When you add a dependency, you specify its lifetime:

- *Transient* objects are created each time they're requested.
- *Scoped* objects are created once per function execution.
- *Singleton* objects are reused across function executions, within the lifetime of the function host.

In this example, the `TelemetryProcessor` and `StateChangeProcessor` objects are declared as transient. This is appropriate for lightweight, stateless services. The `DocumentClient` class, on the other hand, should be a singleton for best performance. For more information, see Performance tips for Azure Cosmos DB and .NET.

If you refer back to the code for the RawTelemetryFunction, you'll see there another dependency that doesn't appear in DI setup code, namely the `TelemetryClient` class that is used to log application metrics. The Functions runtime automatically registers this class into the DI container, so you don't need to register it explicitly.

For more information about DI in Azure Functions, see the following articles:

- Use dependency injection in .NET Azure Functions
- Dependency injection in ASP.NET Core

Passing configuration settings in DI

Sometimes an object must be initialized with some configuration values. Generally, these settings should come from app settings or (in the case of secrets) from Azure Key Vault.

There are two examples in this application. First, the `DocumentClient` class takes a Cosmos DB service endpoint and key. For this object, the application registers a lambda that will be invoked by the DI container. This lambda uses the `IConfiguration` interface to read the configuration values:

```
builder.Services.AddSingleton<IDocumentClient>(ctx => {
    var config = ctx.GetService<IConfiguration>();
    var cosmosDBEndpoint = config.GetValue<string>("CosmosDBEndpoint");
    var cosmosDBKey = config.GetValue<string>("CosmosDBKey");
    return new DocumentClient(new Uri(cosmosDBEndpoint), cosmosDBKey);
});
```

The second example is the `StateChangeProcessor` class. For this object, we use an approach called the options pattern. Here's how it works:

1. Define a class `T` that contains your configuration settings. In this case, the Cosmos DB database name and collection name.

   ```
   public class StateChangeProcessorOptions
   {
       public string COSMOSDB_DATABASE_NAME { get; set; }
       public string COSMOSDB_DATABASE_COL { get; set; }
   }
   ```

2. Add the class `T` as an options class for DI.

   ```
   builder.Services.AddOptions<StateChangeProcessorOptions>()
       .Configure<IConfiguration>((configSection, configuration) =>
       {
           configuration.Bind(configSection);
       });
   ```

3. In the constructor of the class that is being configured, include an `IOptions<T>` parameter.

```
public StateChangeProcessor(IDocumentClient client, IOptions<StateChangeProcessorOptions> options)
```

The DI system will automatically populate the options class with configuration values and pass this to the constructor.

There are several advantages of this approach:

- Decouple the class from the source of the configuration values.
- Easily set up different configuration sources, such as environment variables or JSON configuration files.
- Simplify unit testing.
- Use a strongly typed options class, which is less error prone than just passing in scalar values.

GetStatus function

The other Functions app in this solution implements a simple REST API to get the last-known status of a drone. This function is defined in a class named `GetStatusFunction`. Here is the complete code for the function:

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;
using System.Security.Claims;
using System.Threading.Tasks;

namespace DroneStatusFunctionApp
{
    public static class GetStatusFunction
    {
        public const string GetDeviceStatusRoleName = "GetStatus";

        [FunctionName("GetStatusFunction")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Function, "get", Route = null)]HttpRequest req,
            [CosmosDB(
                databaseName: "%COSMOSDB_DATABASE_NAME%",
                collectionName: "%COSMOSDB_DATABASE_COL%",
                ConnectionStringSetting = "COSMOSDB_CONNECTION_STRING",
                Id = "{Query.deviceId}",
                PartitionKey = "{Query.deviceId}")] dynamic deviceStatus,
            ClaimsPrincipal principal,
            ILogger log)
        {
            log.LogInformation("Processing GetStatus request.");

            if (!principal.IsAuthorizedByRoles(new[] { GetDeviceStatusRoleName }, log))
            {
                return new UnauthorizedResult();
            }

            string deviceId = req.Query["deviceId"];
            if (deviceId == null)
            {
                return new BadRequestObjectResult("Missing DeviceId");
            }

            if (deviceStatus == null)
            {
                return new NotFoundResult();
            }
            else
            {
                return new OkObjectResult(deviceStatus);
            }
        }
    }
}
```

This function uses an HTTP trigger to process an HTTP GET request. The function uses a Cosmos DB input binding to fetch the requested document. One consideration is that this binding will run before the authorization logic is performed inside the function. If an unauthorized user requests a document, the function binding will still fetch the document. Then the authorization code will return a 401, so the user won't see the document. Whether this behavior is acceptable may depend on your requirements. For example, this approach might make it harder to audit data access for sensitive data.

Authentication and authorization

The web app uses Azure AD to authenticate users. Because the app is a single-page application (SPA) running in the browser, the implicit grant flow is appropriate:

1. The web app redirects the user to the identity provider (in this case, Azure AD).
2. The user enters their credentials.
3. The identity provider redirects back to the web app with an access token.
4. The web app sends a request to the web API and includes the access token in the Authorization header.

A Function application can be configured to authenticate users with zero code. For more information, see Authentication and authorization in Azure App Service.

Authorization, on the other hand, generally requires some business logic. Azure AD supports *claims based authentication*. In this model, a user's identity is represented as a set of claims that come from the identity provider. A claim can be any piece of information about the user, such as their name or email address.

The access token contains a subset of user claims. Among these are any application roles that the user is assigned to.

The `principal` parameter of the function is a ClaimsPrincipal object that contains the claims from the access token. Each claim is a key/value pair of claim type and claim value. The application uses these to authorize the request.

The following extension method tests whether a `ClaimsPrincipal` object contains a set of roles. It returns `false` if any of the specified roles is missing. If this method returns false, the function returns HTTP 401 (Unauthorized).

```
namespace DroneStatusFunctionApp
{
    public static class ClaimsPrincipalAuthorizationExtensions
    {
        public static bool IsAuthorizedByRoles(
            this ClaimsPrincipal principal,
            string[] roles,
            ILogger log)
        {
            var principalRoles = new HashSet<string>(principal.Claims.Where(kvp => kvp.Type ==
"roles").Select(kvp => kvp.Value));
            var missingRoles = roles.Where(r => !principalRoles.Contains(r)).ToArray();
            if (missingRoles.Length > 0)
            {
                log.LogWarning("The principal does not have the required {roles}", string.Join(", ",
missingRoles));
                return false;
            }

            return true;
        }
    }
}
```

For more information about authentication and authorization in this application, see the Security considerations section of the reference architecture.

Next steps

Once you get a feel for how this reference solution works, learn best practices and recommendations for similar solutions.

- For a serverless event ingestion solution, see Serverless event processing using Azure Functions.
- For a serverless web app, see Serverless web application on Azure.

Azure Functions is just one Azure compute option. For help with choosing a compute technology, see Choose an Azure compute service for your application.

Related resources

- For in-depth discussion on developing serverless solutions on premises as well as in the cloud, read Serverless apps: Architecture, patterns, and Azure implementation.
- Read more about the Event-driven architecture style.

CI/CD for serverless application frontend on Azure

11/19/2021 • 11 minutes to read • Edit Online

Serverless computing abstracts the servers, infrastructure, and operating systems, allowing developers to focus on application development. A robust *CI/CD* or *Continuous Integration/Continuous Delivery* of such applications allows companies to ship fully tested and integrated software versions within minutes of development. It provides a backbone of modern DevOps environment.

What does CI/CD actually stand for?

- Continuous Integration allows development teams to integrate code changes in a shared repository almost instantaneously. This ability, coupled with automated build and testing before the changes are integrated, ensures that only fully functional application code is available for deployment.
- Continuous Delivery allows changes in the source code, configuration, content, and other artifacts to be delivered to production, and ready to be deployed to end-users, as quickly and safely as possible. The process keeps the code in a *deployable state* at all times. A special case of this is *Continuous Deployment*, which includes actual deployment to end users.

This article discusses a CI/CD pipeline for the web frontend of a serverless reference implementation. This pipeline is developed using Azure services. The web frontend demonstrates a modern web application, with client-side JavaScript, reusable server-side APIs, and pre-built Markup, alternatively called Jamstack. You can find the code in this GitHub repository. The readme describes the steps to download, build, and deploy the application.

The following diagram describes the CI/CD pipeline used in this sample frontend:

This article does not discuss the backend deployment.

Prerequisites

To work with this sample application, make sure you have the following:

- A GitHub account.
- An Azure account. If you don't have one, you can try out a free Azure account.
- An Azure DevOps organization. If you don't have one, you can try out a basic plan, which includes DevOps services such as Azure Pipelines.

Use an online version control system

Version control systems keep track and control changes in your source code. Keeping your source code in an

online version control system allows multiple development teams to collaborate. It is also easier to maintain than a traditional version control on premises. These online systems can be easily integrated with leading CI/CD services. You get the ability to create and maintain the source code in multiple directories, along with build and configuration files, in what is called a *repository*.

The project files for this sample application are kept in GitHub. If you don't have a GitHub account, read this documentation to get started with GitHub repositories.

Automate your build and deploy

Using a CI/CD service such as Azure Pipelines can help you to automate the build and deploy processes. You can create multiple stages in the pipeline, each stage running based on the result of the previous one. The stages can run in either a Windows or Linux container. The script must make sure the tools and environments are set properly in the container. Azure Pipelines can run a variety of build tools, and can work with quite a few online version control systems.

Integrate build tools

Modern build tools can simplify your build process, and provide functionality such as pre-configuration, minification of the JavaScript files, and static site generation. Static site generators can build markup files before they are deployed to the hosting servers, resulting in a fast user experience. You can select from a variety of these tools, based on the type of your application's programming language and platform, as well as additional functionality needed. This article provides a list of popular build tools for a modern application.

The sample is a React application, built using Gatsby.js, which is a static site generator and front-end development framework. These tools can be run locally during development and testing phases, and then integrated with Azure Pipelines for the final deployment.

The sample uses the Gatsby file gatsby-ssr.js for rendering, and gatsby-config.js for site configuration. Gatsby converts all JavaScript files under the `pages` subdirectory of the `src` folder to HTML files. Additional components go in the `components` subdirectory. The sample also uses the gatsby-plugin-typescript plugin that allows using TypeScript for type safety, instead of JavaScript.

For more information about setting up a Gatsby project, see the official Gatsby documentation.

Automate builds

Automating the build process reduces the human errors that can be introduced in manual processes. The file azure-pipelines.yml includes the script for a two-stage automation. The Readme for this project describes the steps required to set up the automation pipeline using Azure Pipelines. The following subsections show how the pipeline stages are configured.

Build stage

Since the Azure Pipeline is integrated with GitHub repository, any changes in the tracked directory of the main branch trigger the first stage of the pipeline, the build stage:

```
trigger:
  batch: true
  branches:
    include:
    - main
  paths:
    include:
    - src/ClientApp
```

The following snippet illustrates the start of the build stage, which starts an Ubuntu container to run this stage.

```
stages:
- stage: Build
  jobs:
  - job: WebsiteBuild
    displayName: Build Fabrikam Drone Status app
    pool:
      vmImage: 'Ubuntu-16.04'
    continueOnError: false
steps:
```

This is followed by *tasks* and *scripts* required to successfully build the project. These include the following:

- Installing Node.js and setting up environment variables,
- Installing and running Gatsby.js that builds the static website:

```
    - script: |
        cd src/ClientApp
        npm install
        npx gatsby build
      displayName: 'gatsby build'
```

- installing and running a compression tool named *brotli*, to compress the built files before deployment:

```
    - script: |
        cd src/ClientApp/public
        sudo apt-get install brotli --install-suggests --no-install-recommends -q --assume-yes
        for f in $(find . -type f \( -iname '*.html' -o -iname '*.map' -o -iname '*.js' -o -iname
'*.json' \)); do brotli $f -Z -j -f -v && mv ${f}.br $f; done
      displayName: 'enable compression at origin level'
```

- Computing the version of the current build for cache management,
- Publishing the built files for use by the deploy stage:

```
    - task: PublishPipelineArtifact@1
      inputs:
        targetPath: 'src/ClientApp/public'
        artifactName: 'drop'
```

A successful completion of the build stage tears down the Ubuntu environment, and triggers the deploy stage in the pipeline.

Deploy stage

The deploy stage runs in a new Ubuntu container:

```
- stage: Deploy
  jobs:
  - deployment: WebsiteDeploy
    displayName: Deploy Fabrikam Drone Status app
    pool:
      vmImage: 'Ubuntu-16.04'
    environment: 'fabrikamdronestatus-prod'
    strategy:
      runOnce:
        deploy:
          steps:
```

This stage includes various deployment tasks and scripts to:

- Download the build artifacts to the container (which happens automatically as a consequence of using `PublishPipelineArtifact` in the build stage),
- Record the build release version, and update in the GitHub repository,
- Upload the website files to Blob Storage, in a new folder corresponding to the new version, and
- Change the CDN to point to this new folder.

The last two steps together replicate a cache purge, since older folders are no longer accessible by the CDN edge servers. The following snippet shows how this is achieved:

```
       - script: |
              az login --service-principal -u $(azureArmClientId) -p $(azureArmClientSecret) --tenant
$(azureArmTenantId)
              # upload content to container versioned folder
              az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --destination
"\$web\$(releaseSemVer)" --account-name $(azureStorageAccountName) --content-encoding br --pattern "*.html"
--content-type "text/html"
              az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --destination
"\$web\$(releaseSemVer)" --account-name $(azureStorageAccountName) --content-encoding br --pattern "*.js" --
content-type "application/javascript"
              az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --destination
"\$web\$(releaseSemVer)" --account-name $(azureStorageAccountName) --content-encoding br --pattern
"*.js.map" --content-type "application/octet-stream"
              az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --destination
"\$web\$(releaseSemVer)" --account-name $(azureStorageAccountName) --content-encoding br --pattern "*.json"
--content-type "application/json"
              az storage blob upload-batch -s "$(Pipeline.Workspace)/drop" --destination
"\$web\$(releaseSemVer)" --account-name $(azureStorageAccountName) --pattern "*.txt" --content-type
"text/plain"
              # target new version
              az cdn endpoint update --resource-group $(azureResourceGroup) --profile-name $(azureCdnName) -
-name $(azureCdnName) --origin-path '/$(releaseSemVer)'
              AZURE_CDN_ENDPOINT_HOSTNAME=$(az cdn endpoint show --resource-group $(azureResourceGroup) --
name $(azureCdnName) --profile-name $(azureCdnName) --query hostName -o tsv)
              echo "Azure CDN endpoint host ${AZURE_CDN_ENDPOINT_HOSTNAME}"
              echo '##vso[task.setvariable variable=azureCndEndpointHost]'$AZURE_CDN_ENDPOINT_HOSTNAME
            displayName: 'upload to Azure Storage static website hosting and purge Azure CDN endpoint'
```

Atomic deploys

Atomic deployment ensures that the users of your website or application always get the content corresponding to the same version.

In the sample CI/CD pipeline, the website contents are deployed to the Blob storage, which acts as the origin server for the Azure CDN. If the files are updated in the same *root folder* in the blob, the website will be served inconsistently. Uploading to a new versioned folder as shown in the preceding section solves this problem. The users either get *all or nothing* of the new successful build, since the CDN points to the new folder as the origin, only after all files are successfully updated.

The advantages of this approach are as follows:

- Since new content is not available to users until the CDN points to the new origin folder, it results in an atomic deployment.
- You can easily roll back to an older version of the website if necessary.
- Since the origin can host multiple versions of the website side by side, you can fine-tune the deployment by using techniques such as allowing preview to certain users before wider availability.

Host and distribute using the cloud

A content delivery network (CDN) is a set of distributed servers that speed up the content delivery to users over a vast geographical area. Every user gets the content from the server nearest to them. The CDN accesses this content from an *origin* server, and caches it to *edge* servers at strategic locations. The sample CI/CD in this article uses Azure CDN, pointing to website content hosted on Azure Blob Storage as the origin server. The Blob storage is configured for static website hosting. For a quick guide on how to use Azure CDN with Azure Blob Storage, read Integrate an Azure storage account with Azure CDN.

The following are some strategies that can improve the CDN performance.

Compress the content

Compressing the files before serving improves file transfer speed and increases page-load performance for the end users.

There are two ways to do this:

1. **On the fly in the CDN edge servers**. This improves bandwidth consumption by a significant percentage, making the website considerably faster than without compression. It is relatively easy to enable this type of compression on Azure CDN. Since it's controlled by the CDN, you cannot choose or configure the compression tool. Use this compression if website performance is not a critical concern.

2. **Pre-compressing before reaching the CDN**, either on your origin server, or before reaching the origin. Pre-compressing further improves the run time performance of your website, since it's done before being fetched by the CDN. The sample CI/CD pipeline compresses the built files in the deployment stage, using brotli, as shown in the build section above. The advantages of using pre-compression are as follows:

    a. You can use any compression tool that you're comfortable with, and further fine-tune the compression. CDNs may have limitations on the tools they use.
    b. Some CDNs limit file sizes that can be compressed on the fly, causing a performance hit for larger

files. Pre-compression sets no limits on the file size.

c. Pre-compressing in the deployment pipeline results in less storage required at the origin.
d. This is faster and more efficient than the CDN compression.

For more information, see Improve performance by compressing files in Azure CDN.

Dynamic site acceleration

Using CDN is optimal for static content, which can be safely cached at the edge servers. However, dynamic web sites require the server to generate content based on user response. This type of content cannot be cached on the edge, requiring a more involved end-to-end solution that can speed up the content delivery. Dynamic site acceleration by Azure CDN is one such solution that measurably improves the performance of dynamic web pages.

This sample enables dynamic site acceleration as shown in this section of the readme.

Manage website cache

CDNs use caching to improve their performance. Configuring and managing this cache becomes an integral part of your deployment pipeline. The Azure CDN documentation shows several ways to do this. You can set caching rules on your CDN, as well as configure the time-to-live for the content in the origin server. Static web content can be cached for a long duration, since it may not change too much over time. This reduces the overhead of accessing the single origin server for every user request. For more information, see How caching works.

Cache purge

The CDN caches the website files at the edge servers until their time-to-live expires. These are updated for a new client request, only after their time-to-live has expired. Purging your CDN cache is required to guarantee that every user gets the latest live website files, especially if the deployment happens at the same CDN origin folder. Azure CDN allows you to purge the cache from the Azure portal.

A better approach is to invalidate this cache by using versioning during deployment. An explicit cache purge or expiration usually causes the CDN to retrieve newer versions of the web content. However, since the CDN always points to the latest version of the deployment, it improves caching in the following manner:

1. The CDN validates the index.html against the origin, for every new website instance.
2. Except for the index.html and 404.html, all other website files are fingerprinted and cached for a year. This is based on the assumption that resources such as images and videos, do not need frequent changes. If these files are updated and rebuilt, their names are updated by a new fingerprint GUID. This results in an update to the index.html with an updated reference to the changed resource file. The CDN then retrieves the updated index.html, and since it does not find the reference resource file in its cache, it also retrieves the changed resource files.

This ensures that the CDN always gets new updated files, and removes the need to purge the cache for a new build.

Next steps

- Now that you understand the basics, follow this readme to set up and execute the CI/CD pipeline.
- Learn best practices for using content delivery networks (CDNs)

Monitor serverless event processing

11/19/2021 • 8 minutes to read • Edit Online

This article provides guidance on monitoring serverless event-driven architectures.

Monitoring provides insight into the behavior and health of your systems. It helps you build a holistic view of the environment, retrieve historic trends, correlate diverse factors, and measure changes in performance, consumption, or error rate. You can use monitoring to define alerts when conditions occur that could impact the quality of your service, or when conditions of particular interest to your specific environment arise.

This article demonstrates using Azure Monitor to monitor a serverless application built using Event Hubs and Azure Functions. It discusses useful metrics to monitor, describes how to integrate with Application Insights and capture custom metrics, and provides code samples.

Assumptions

This article assumes you have an architecture like the one described in the Serverless event processing reference architecture. Basically:

- Events arrive at Azure Event Hubs.
- A Function App is triggered to handle the event.
- Azure Monitor is available for use with your architecture.

Metrics from Azure Monitor

First we need to decide which metrics will be needed before we can begin to formulate useful insights about the architecture. Each resource performs different tasks, and in turn generates different metrics.

These metrics from Event Hub will be of interest to capture useful insights:

- Incoming requests
- Outgoing requests
- Throttled requests
- Successful requests
- Incoming messages
- Outgoing messages
- Captured messages
- Incoming bytes
- Outgoing bytes
- Captured bytes
- User errors

Similarly, these metrics from Azure Functions will be of interest to capture useful insights:

- Function execution count
- Connections
- Data in
- Data out
- HTTP server errors
- Requests

- Requests in application queue
- Response time

Using diagnostics logging to capture insights

When analyzed together, the above metrics can be used to formulate and capture the following insights:

- Rate of requests processed by Event Hubs
- Rate of requests processed by Azure Functions
- Total Event Hub throughput
- User errors
- Duration of Azure Functions
- End-to-end latency
- Latency at each stage
- Number of messages lost
- Number of messages processed more than once

To ensure that Event Hubs captures the necessary metrics, we must first enable diagnostic logs (which are disabled by default). We must then select which logs are desired and configure the correct Log Analytics workspace as the destination for them.

The log and metric categories that we are interested in are:

- OperationalLogs
- AutoScaleLogs
- KafkaCoordinatorLogs *(for Apache Kafka workloads)*
- KafkaUserErrorLogs *(for Apache Kafka workloads)*
- EventHubVNetConnectionEvent
- AllMetrics

Azure documentation provides instructions on how to Set up diagnostic logs for an Azure event hub. The following screenshot shows an example *Diagnostic setting* configuration panel with the correct log and metric categories selected, and a Log Analytics workspace set as the destination. (If an external system is being used to analyze the logs, the option to *Stream to an event hub* can be used instead.)

NOTE

In order to utilize log diagnostics to capture insights, you should create event hubs in different namespaces. This is because of a constraint in Azure.

The Event Hubs set in a given Event Hubs namespace is represented in Azure Monitor metrics under a dimension called `EntityName`. In the Azure portal, data for a specific event hub normally can be viewed on that instance of Azure Monitor. But when the metrics data is routed to the log diagnostics, there is currently no way to view data per event hub by filtering on the `EntityName` dimension.

As a workaround, creating event hubs in different namespaces helps make it possible to locate metrics for a specific hub.

Using Application Insights

You can enable Application Insights to capture metrics and custom telemetry from Azure Functions. This allows you to define analytics that suit your own purposes, providing another way to get important insights for the serverless event processing scenario.

This screenshot shows an example listing of custom metrics and telemetry within Application Insights:

timestamp [UTC]	message	severityLevel	itemType	customDimensions
11/10/2020, 2:43:28.754 AM	Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Rea...	1	trace	{"prop__{OriginalFormat}":"Exec
11/10/2020, 2:43:28.754 AM	Trigger Details: PartionId: 14, Offset: 25800738240, EnqueueTimeUtc:...	1	trace	{"prop__{OriginalFormat}":"Trigg
11/10/2020, 2:43:28.754 AM	Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Rea...	1	trace	{"prop__{OriginalFormat}":"Exec
11/10/2020, 2:43:28.754 AM	Trigger Details: PartionId: 14, Offset: 25800738240, EnqueueTimeUtc:...	1	trace	{"prop__{OriginalFormat}":"Trigg
11/10/2020, 2:43:28.753 AM	Executing 'EventHubStandardTriggeredFunction_AsSingleEvent' (Rea...	1	trace	{"prop__{OriginalFormat}":"Exec
11/10/2020, 2:43:28.716 AM	Executed 'EventHubStandardTriggeredFunction_AsSingleEvent' (Succ...	1	trace	{"prop__{OriginalFormat}":"Exec
11/10/2020, 2:43:28.716 AM	Read Payload from message: {"Body":[123,34,107,101,121,34,58,34,97,...	1	trace	{"prop__{OriginalFormat}":"Read
11/10/2020, 2:43:28.715 AM	Executed 'EventHubStandardTriggeredFunction_AsSingleEvent' (Succ...	1	trace	{"prop__{OriginalFormat}":"Exec
11/10/2020, 2:43:28.715 AM	Read Payload from message: {"Body":[123,34,107,101,121,34,58,34,53,...	1	trace	{"prop__{OriginalFormat}":"Read

Default custom metrics

In Application Insights, custom metrics for Azure Functions are stored in the `customMetrics` table. It includes the following values, spanned over a timeline for different function instances:

- `AvgDurationMs`
- `MaxDurationMs`
- `MinDurationMs`
- `Successes`
- `Failures`
- `SuccessRate`
- `Count`

These metrics can be used to efficiently calculate the aggregated averages across the multiple function instances that are invoked in a run.

This screenshot shows what these default custom metrics look like when viewed in Application Insights:

timestamp [UTC]	name	value	valueCount	valueSum	valueMin
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent Count	1,000	1	1,000	1,000
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent AvgDurationMs	84.757	1	84.757	84.757
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent MaxDurationMs	276.234	1	276.234	276.234
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent MinDurationMs	11.744	1	11.744	11.744
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent Successes	1,000	1	1,000	1,000
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent Failures	0	1	0	0
11/10/2020, 2:36:38.242 AM	EventHubStandardTriggeredFunction_AsSingleEvent SuccessRate	100	1	100	100

Custom messages

Custom messages logged in the Azure Function code (using the `ILogger`) are obtained from the Application Insights `traces` table.

The `traces` table has the following important properties (among others):

- `timestamp`
- `cloud_RoleInstance`
- `operation_Id`
- `operation_Name`
- `message`

Here is an example of what a custom message might look like in the Application Insights interface:

11/10/2020, 2:47:27.960 AM	Read Payload from message: {"Body":[123,34,107,101,121,34,58,34,52,98,100,54,57,48,100,52,45,101,98,54,54,45,52,...	1
timestamp [UTC]	2020-11-10T02:47:27.9609674Z	
message	Read Payload from message: {"Body":[123,34,107,101,121,34,58,34,52,98,100,54,57,48,100,52,45,101,98,54,54,45,52,54,50,101,4	
severityLevel	1	
itemType	trace	
customDimensions	{"prop__{OriginalFormat}":"Read Payload from message: {\"Body\":[123,34,107,101,121,34,58,34,52,98,100,54,57,48,100,52,45,1	
operation_Name	EventHubStandardTriggeredFunction_AsSingleEvent	
operation_Id	6b66dca58934a04cabdb26acd4e29a0c	
operation_ParentId	f67b02405e1cf146	
client_Type	PC	
client_IP	0.0.0.0	
cloud_RoleName	samltbkendwinyfx-dotnet	
cloud_RoleInstance	b637eff58cea882e5c19a73e4e44cca8088f890a8b41921aa654b05c5e871721	
appId	0e0902f4-e3fd-4624-9dec-64bb06751b50	
appName	samltbkendwinyfx	
iKey	47171669-fd25-42a5-b7b3-552d9168743d	
sdkVersion	azurefunctions: 3.0.14916.0	
itemId	12271dcd-22ff-11eb-b726-fff2d030959d	
itemCount	1	

If the incoming Event Hub message or `EventData[]` is logged as a part of this custom `ILogger` message, then that is also made available in Application Insights. This can be very useful.

For our serverless event processing scenario, we log the JSON serialized message body that's received from the event hub. This allows us to capture the raw byte array, along with `SystemProperties` like `x-opt-sequence-number`, `x-opt-offset`, and `x-opt-enqueued-time`. To determine when each message was received by the Event Hub, the `x-opt-enqueued-time` property is used.

Sample query:

```
traces
| where timestamp between(min_t .. max_t)
| where message contains "Body"
| extend m = parse_json(message))
| project timestamp = todatetime(m.SystemProperties.["x-opt-enqueued-time"])
```

The sample query would return a message similar to the following example result, which gets logged by default in Application Insights. The properties of the `Trigger Details` can be used to locate and capture additional insights around messages received per `PartitionId`, `Offset`, and `SequenceNumber`.

Example result of the sample query:

```
"message": Trigger Details: PartitionId: 26, Offset: 17194119200, EnqueueTimeUtc: 2020-11-
03T02:14:01.7740000Z, SequenceNumber: 843572, Count: 10,
```

WARNING

The library for Azure Java Functions currently has an issue that prevents access to the `PartitionID` and the `PartitionContext` when using `EventHubTrigger`. Learn more in this GitHub issue report.

Tracking message flow using a transaction ID with Application Insights

In Application Insights, we can view all the telemetry related to a particular transaction by doing a Transaction

search query on the transaction's `Operation Id` value. This can be especially useful for capturing the percentile values of average times for messages as the transaction moves through the event stream pipeline.

The following screenshot shows an example Transaction search in the Application Insights interface. The desired `Operation ID` is entered in the query field, identified with a magnifying glass icon (and shown here outlined in a red box). At the bottom of the main pane, the `Results` tab shows matching events in sequential order. In each event entry, the `Operation ID` value is highlighted in dark blue for easy verification.

A query generated for a specific operation ID will look like the following. Note that the `Operation ID` GUID is specified in the third line's `where * has` clause. This example further narrows the query between two different `datetimes`.

```
union isfuzzy=true availabilityResults, requests, exceptions, pageViews, traces, customEvents, dependencies
| where timestamp > datetime("2020-10-09T06:58:40.024Z") and timestamp < datetime("2020-11-11T06:58:40.024Z")
| where * has "1c8c9d7073a00e4bbdcc8f2e6570e46"
| order by timestamp desc
| take 100
```

Here is a screenshot of what the query and its matching results might look like in the Application Insights interface:

Capture custom metrics from Azure Functions

.NET functions

Structured logging is used in the .NET Azure functions for capturing custom dimensions in the Application Insights traces table. These custom dimensions can then be used for querying data.

As an example, here is the log statement in the .NET `TransformingFunction`:

```
log.LogInformation("TransformingFunction: Processed sensorDataJson={sensorDataJson}, " +
    "partitionId={partitionId}, offset={offset} at {enqueuedTimeUtc}, " +
    "inputEH_enqueuedTime={inputEH_enqueuedTime}, processedTime={processedTime}, " +
    "transformingLatencyInMs={transformingLatencyInMs}, processingLatencyInMs={processingLatencyInMs}",
    sensorDataJson,
    partitionId,
    offset,
    enqueuedTimeUtc,
    inputEH_enqueuedTime,
    processedTime,
    transformingLatency,
    processingLatency);
```

The resulting logs created on Application Insights contain the above parameters as custom dimensions, as shown in this screenshot:

These logs can be queried as follows:

```
traces
| where timestamp between(min_t .. max_t)
// Function name should be of the function consuming from the Event Hub of interest
| where operation_Name == "{Function_Name}"
| where message has "{Function_Name}: Processed"
| project timestamp = todatetime(customDimensions.prop__enqueuedTimeUtc)
```

> **NOTE**
>
> In order to make sure we do not affect performance in these tests, we have turned on the sampling settings of Azure Function logs for Application Insights using the `host.json` file as shown below. This means that all statistics captured from logging are considered to be average values and not actual counts.

host.json example:

```
"logging": {
    "applicationInsights": {
        "samplingExcludedTypes": "Request",
        "samplingSettings": {
            "isEnabled": true
        }
    }
}
```

Java functions

Currently, structured logging isn't supported in Java Azure functions for capturing custom dimensions in the Application Insights traces table.

As an example, here is the log statement in the Java `TransformingFunction`:

```
LoggingUtilities.logSuccessInfo(
    context.getLogger(),
    "TransformingFunction",
    "SuccessInfo",
    offset,
    processedTimeString,
    dateformatter.format(enqueuedTime),
    transformingLatency
);
```

The resulting logs created on Application Insights contain the above parameters in the message as shown below:

These logs can be queried as follows:

```
traces
| where timestamp between(min_t .. max_t)
// Function name should be of the function consuming from the Event Hub of interest
| where operation_Name in ("{Function name}") and message contains "SuccessInfo"
| project timestamp = todatetime(tostring(parse_json(message).enqueuedTime))
```

> **NOTE**
> In order to make sure we do not affect performance in these tests, we have turned on the sampling settings of Azure Function logs for Application Insights using the `host.json` file as shown below. This means that all statistics captured from logging are considered to be average values and not actual counts.

host.json example:

```
"logging": {
    "applicationInsights": {
        "samplingExcludedTypes": "Request",
        "samplingSettings": {
            "isEnabled": true
        }
    }
}
```

Related resources

- Serverless event processing is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- De-batching and filtering in serverless event processing with Event Hubs describes in more detail how these portions of the reference architecture work.
- Private link scenario in event stream processing is a solution idea for implementing a similar architecture in a virtual network (VNet) with private endpoints, in order to enhance security.
- Azure Kubernetes in event stream processing describes a variation of a serverless event-driven architecture running on Azure Kubernetes with KEDA scaler.

Serverless Functions app operations

11/19/2021 • 3 minutes to read • Edit Online

This article describes Azure operations considerations for serverless Functions applications. To support Functions apps, operations personnel need to:

- Understand and implement hosting configurations.
- Future-proof scalability by automating infrastructure provisioning.
- Maintain business continuity by meeting availability and disaster recovery requirements.

Planning

To plan operations, understand your workloads and their requirements, then design and configure the best options for the requirements.

Choose a hosting option

The Azure Functions Runtime provides flexibility in hosting. Use the hosting plan comparison table to determine the best choice for your requirements.

- Azure Functions hosting plans

  Each Azure Functions project deploys and runs in its own Functions app, which is the unit of scale and cost. The three hosting plans available for Azure Functions are the Consumption plan, Premium plan, and Dedicated (App Service) plan. The hosting plan determines scaling behavior, available resources, and support for advanced features like virtual network connectivity.

- Azure Kubernetes Service (AKS)

  Kubernetes-based Functions provides the Functions Runtime in a Docker container with event-driven scaling through Kubernetes-based Event Driven Autoscaling (KEDA).

For more information about hosting plans, see:

- Azure Functions scale and hosting
- Consumption plan
- Premium plan
- Dedicated (App Service) plan
- Azure Functions on Kubernetes with KEDA
- Azure subscription and service limits, quotas, and constraints

Understand scaling

The serverless Consumption and Premium hosting plans *scale* automatically, adding and removing Azure Functions host instances based on the number of incoming events. Scaling can vary on several dimensions, and behave differently based on plan, trigger, and code language.

For more information about scaling, see:

- Understand scaling behaviors
- Scalability best practices

Understand and address cold starts

If the number of host instances scales down to zero, the next request has the added latency of restarting the

Function app, called a *cold start*. Cold start is a large discussion point for serverless architectures, and a point of ambiguity for Azure Functions.

The Premium hosting plan prevents cold starts by keeping some instances warm. Reducing dependencies and using asynchronous operations in the Functions app also minimizes the impact of cold starts. However, availability requirements may require running the app in a Dedicated hosting plan with *Always on* enabled. The Dedicated plan uses dedicated virtual machines (VMs), so is not serverless.

For more information about cold start, see Understanding serverless cold start.

Identify storage considerations

Every Azure Functions app relies on Azure Storage for operations such as managing triggers and logging function executions. When creating a Functions app, you must create or link to a general-purpose Azure Storage account that supports Blob, Queue, and Table storage. For more information, see Storage considerations for Azure Functions.

Identify network design considerations

Networking options let the Functions app restrict access, or access resources without using internet-routable addresses. The hosting plans offer different levels of network isolation. Choose the option that best meets your network isolation requirements. For more information, see Azure Functions networking options.

Production

To prepare the application for production, make sure you can easily redeploy the hosting plan, and apply scale-out rules.

Automate hosting plan provisioning

With infrastructure as code, you can automate infrastructure provisioning. Automatic provisioning provides more resiliency during disasters, and more agility to quickly redeploy the infrastructure as needed.

For more information on automated provisioning, see:

- Automate resource deployment for your function app in Azure Functions
- Terraform - Manages a Function App

Configure scale out options

Autoscale provides the right amount of running resources to handle application load. Autoscale adds resources to handle increases in load, and saves money by removing resources that are idle.

For more information about autoscale options, see:

- Premium Plan settings
- App Service Plan settings

Optimization

When the application is in production, make sure that:

- The hosting plan can scale to meet application demands.
- There's a plan for business continuity, availability, and disaster recovery.
- You can monitor hosting and application health and receive alerts.

Implement availability requirements

Azure Functions run in a specific region. To get higher availability, you can deploy the same Functions app to multiple regions. In multiple regions, Functions can run in the *active-active* or *active-passive* availability pattern.

For more information about Azure Functions availability and disaster recovery, see:

- Azure Functions geo-disaster recovery
- Disaster recovery and geo-distribution in Azure Durable Functions

Monitoring logging, application monitoring, and alerting

Application Insights and logs in Azure Monitor automatically collect log, performance, and error data and detect performance anomalies. Azure Monitor includes powerful analytics tools to help diagnose issues and understand function use. Application Insights help you continuously improve performance and usability.

For more information about monitoring and analyzing Azure Functions performance, see:

- Monitor Azure Functions
- Monitor Azure Functions with Azure Monitor logs
- Application Insights for Azure Functions supported features

Next steps

- Serverless application development and deployment
- Azure Functions app security

Serverless Functions security

11/19/2021 • 5 minutes to read • Edit Online

This article describes Azure services and activities security personnel can implement for serverless Functions. These guidelines and resources help develop secure code and deploy secure applications to the cloud.

Planning

The primary goals of a secure serverless Azure Functions application environment are to protect running applications, quickly identify and address security issues, and prevent future similar issues.

The OWASP Serverless Top 10 describes the most common serverless application security vulnerabilities, and provides basic techniques to identify and protect against them.

In many ways, planning for secure development, deployment, and operation of serverless functions is much the same as for any web-based or cloud hosted application. Azure App Service provides the hosting infrastructure for your function apps. Securing Azure Functions article provides security strategies for running your function code, and how App Service can help you secure your functions.

For more information about Azure security, best practices, and shared responsibilities, see:

- Security in Azure App Service
- Built-in security controls
- Secure development best practices on Azure.
- Security best practices for Azure solutions (PDF report)
- Shared responsibilities for cloud computing (PDF report)

Deployment

To prepare serverless Functions applications for production, security personnel should:

- Conduct regular code reviews to identify code and library vulnerabilities.
- Define resource permissions that Functions needs to execute.
- Configure network security rules for inbound and outbound communication.
- Identify and classify sensitive data access.

The Azure Security Baseline for Azure Functions article contains more recommendations that will help you improve the security posture of your deployment.

Keep code secure

Find security vulnerabilities and errors in code and manage security vulnerabilities in projects and dependencies.

For more information, see:

- GitHub - Finding security vulnerabilities and errors in your code
- GitHub - Managing security vulnerabilities in your project
- GitHub - Managing vulnerabilities in your project's dependencies

Perform input validation

Different event sources like Blob storage, Cosmos DB NoSQL databases, event hubs, queues, or Graph events can trigger serverless Functions. Injections aren't strictly limited to inputs coming directly from the API calls.

Functions may consume other input from the possible event sources.

In general, don't trust input or make any assumptions about its validity. Always use safe APIs that sanitize or validate the input. If possible, use APIs that bind or parameterize variables, like using prepared statements for SQL queries.

For more information, see:

- Azure Functions Input Validation with FluentValidation
- Security Frame: Input Validation Mitigations
- HTTP Trigger Function Request Validation
- How to validate request for Azure Functions

Secure HTTP endpoints for development, testing, and production

Azure Functions lets you use keys to make it harder to access your HTTP function endpoints. To fully secure your function endpoints in production, consider implementing one of the following Function app-level security options:

- Turn on App Service authentication and authorization for your Functions app. See Authorization keys.
- Use Azure API Management (APIM) to authenticate requests. See Import an Azure Function App as an API in Azure API Management.
- Deploy your Functions app to an Azure App Service Environment (ASE).
- Use an App Service Plan that restricts access, and implement Azure Front Door + WAF to handle your incoming requests. See Create a Front Door for a highly available global web application.

For more information, see Secure an HTTP endpoint in production.

Set up Azure role-based access control (Azure RBAC)

Azure role-based access control (Azure RBAC) has several Azure built-in roles that you can assign to users, groups, service principals, and managed identities to control access to Azure resources. If the built-in roles don't meet your organization's needs, you can create your own Azure custom roles.

Review each Functions app before deployment to identify excessive permissions. Carefully examine functions to apply "least privilege" permissions, giving each function only what it needs to successfully execute.

Use Azure RBAC to assign permissions to users, groups, and applications at a certain scope. The scope of a role assignment can be a subscription, a resource group, or a single resource. Avoid using wildcards whenever possible.

For more information about Azure RBAC, see:

- What is Azure role-based access control (Azure RBAC)?
- Azure built-in roles
- Azure custom roles

Use managed identities and key vaults

A common challenge when building cloud applications is how to manage credentials for authenticating to cloud services in your code. Credentials should never appear in application code, developer workstations, or source control. Instead, use a key vault to store and retrieve keys and credentials. Azure Key Vault provides a way to securely store credentials, secrets, and other keys. The code authenticates to Key Vault to retrieve the credentials.

For more information, see Use Key Vault references for App Service and Azure Functions.

Managed identities let Functions apps access resources like key vaults and storage accounts without requiring specific access keys or connection strings. A full audit trail in the logs displays which identities execute requests to resources. Use Azure RBAC and managed identities to granularly control exactly what resources Azure

Functions applications can access.

For more information, see:

- What are managed identities for Azure resources?
- How to use managed identities for App Service and Azure Functions

Use shared access signature (SAS) tokens to limit access to resources

A *shared access signature (SAS)* provides secure delegated access to resources in your storage account, without compromising the security of your data. With a SAS, you have granular control over how a client can access your data. You can control what resources the client may access, what permissions they have on those resources, and how long the SAS is valid, among other parameters.

For more information, see Grant limited access to Azure Storage resources using shared access signatures (SAS).

Secure Blob storage

Identify and classify sensitive data, and minimize sensitive data storage to only what is necessary. For sensitive data storage, add multi-factor authentication and data encryption in transit and at rest. Grant limited access to Azure Storage resources using SAS tokens.

For more information, see Security recommendations for Blob storage.

Optimization

Once an application is in production, security personnel can help optimize workflow and prepare for scaling.

Use Microsoft Defender for Cloud and apply security recommendations

Microsoft Defender for Cloud is a security scanning solution for your application that identifies potential security vulnerabilities and creates recommendations. The recommendations guide you to configure needed controls to harden and protect your resources.

For more information, see:

- Protect your applications with Microsoft Defender for Cloud
- Defender for Cloud app recommendations

Enforce application governance policies

Apply centralized, consistent enforcements and safeguards to your application at scale. For more information, see Azure Policy built-in policy definitions.

Next steps

- Serverless application development and deployment
- Azure Functions app operations

Resilient Event Hubs and Functions design

11/19/2021 • 11 minutes to read • Edit Online

Error handling, designing for idempotency and managing retry behavior are a few of the critical measures you can take to ensure Event Hubs triggered functions are resilient and capable of handling large volumes of data. This article covers these crucial concepts and makes recommendations for serverless event-streaming solutions.

Azure provides three main messaging services that can be used with Azure Functions to support a wide range of unique, event-driven scenarios. Because of its partitioned consumer model and ability to ingest data at a high rate, Azure Event Hubs is commonly used for event streaming and big data scenarios. For a detailed comparison of Azure messaging services, see Choose between Azure messaging services - Event Grid, Event Hubs, and Service Bus.

Streaming benefits and challenges

Understanding the benefits and drawbacks of streams helps you appreciate how a service like Event Hubs operates. You also need this context when making impactful architectural decisions, troubleshooting issues, and optimizing for performance. Consider the following key concepts about solutions featuring both Event Hubs and Functions:

- **Streams are not queues:** Event Hubs, Kafka, and other similar offerings that are built on the partitioned consumer model don't intrinsically support some of the principal features in a message broker like Service Bus. Perhaps the biggest indicator of this is the fact that reads are **non-destructive**. This means that the data that is read by the Functions host isn't deleted afterwards. Instead, messages are immutable and remain for other consumers to read, including potentially the same customer reading it again. For this reason, solutions that implement patterns such as competing consumers are better suited for a traditional message broker.
- **Missing inherit dead-letter support:** A dead-letter channel is not a native feature in Event Hubs or Kafka. Often, the *concept* of dead-lettering is integrated into a streaming solution to account for data that cannot be processed. This functionality is intentionally not an innate element in Event Hubs and is only added on the consumer side to manufacture a similar behavior or effect. If you need dead-letter support, you should potentially review your choice of streaming message service.
- **A unit of work is a partition:** In a traditional message broker, a unit of work is a single message. In a streaming solution, a partition is often considered the unit of work. If each event in an event hub is regarded as a discrete message that requires it to be treated like an order processing operation or financial transaction, it's most likely an indication of the wrong messaging service being used.
- **No server-side filtering:** One of the reasons Event Hubs is capable of tremendous scale and throughput is due to the low overhead on the service itself. Features like server-side filtering, indexes, and cross-broker coordination aren't part of the architecture of Event Hubs. Functions are occasionally used to filter events by routing them to other Event Hubs based on the contents in the body or header. This approach is common in event streaming but comes with the caveat that each event is read and evaluated by the initial function.
- **Every reader must read all data:** Since server-side filtering is unavailable, a consumer sequentially reads all the data in a partition. This includes data that may not be relevant or could even be malformed. There are several options and even strategies that can be used to compensate for these challenges that will be covered later in this section.

These significant design decisions allow Event Hubs to do what it does best: support a significant influx of events

and provide a robust and resilient service for consumers to read from. Each consumer application is tasked with the responsibility of maintaining their own, client-side offsets or cursor to those events. The low overhead makes Event Hubs an affordable and powerful option for event streaming.

Idempotency

One of the core tenets of Azure Event Hubs is the concept of at-least once delivery. This approach ensures that events will always be delivered. It also means that events can be received more than once, even repeatedly, by consumers such as a function. For this reason, it's important that an event hub triggered function supports the idempotent consumer pattern.

Working under the assumption of at-least once delivery, especially within the context of an event-driven architecture, is a responsible approach for reliably processing events. Your function must be idempotent so that the outcome of processing the same event multiple times is the same as processing it once.

Duplicate events

There are several different scenarios that could result in duplicate events being delivered to a function:

- **Checkpointing:** If the Azure Functions host crashes, or the threshold set for the batch checkpoint frequency is not met, a checkpoint will not be created. As a result, the offset for the consumer is not advanced and the next time the function is invoked, it will resume from the last checkpoint. It is important to note that checkpointing occurs at the partition level for each consumer.
- **Duplicate events published:** There are many techniques that could alleviate the possibility of the same event being published to a stream, however, it's still the responsibility of the consumer to idempotently handle duplicates.
- **Missing acknowledgments:** In some situations, an outgoing request to a service may be successful, however, an acknowledgment (ACK) from the service is never received. This might result in the perception that the outgoing call failed and initiate a series or retries or other outcomes from the function. In the end, duplicate events could be published, or a checkpoint is not created.

Deduplication techniques

Designing your functions for identical input should be the default approach taken when paired with the Event Hub trigger binding. You should consider the following techniques:

- **Looking for duplicates:** Before processing, take the necessary steps to validate that the event should be processed. In some cases, this will require an investigation to confirm that it is still valid. It could also be possible that handling the event is no longer necessary due to data freshness or logic that invalidates the event.
- **Design events for idempotency:** By providing additional information within the payload of the event, it may be possible to ensure that processing it multiple times will not have any detrimental effects. Take the example of an event that includes an amount to withdrawal from a bank account. If not handled responsibly, it is possible that it could decrement the balance of an account multiple times. However, if the same event includes the updated balance to the account, it could be used to perform an upsert operation to the bank account balance. This event-carried state transfer approach occasionally requires coordination between producers and consumers and should be used when it makes sense to participating services.

Error handling and retries

Error handling and retries are a few of the most important qualities of distributed, event-driven applications, and Functions are no exception. For event streaming solutions, the need for proper error handling support is crucial, as thousands of events can quickly turn into an equal number of errors if they are not handled correctly.

Error handling guidance

Without error handling, it can be tricky to implement retries, detect runtime exceptions, and investigate issues. Every function should have at least some level or error handling. A few recommended guidelines are:

- **Use Application Insights:** Enable and use Application Insights to log errors and monitor the health of your functions. Be mindful of the configurable sampling options for scenarios that process a high volume of events.
- **Add structured error handling:** Apply the appropriate error handling constructs for each programming language to catch, log, and detect anticipated and unhandled exceptions in your function code. For instance, use a try/catch block in C#, Java and JavaScript and take advantage of the try and except blocks in Python to handle exceptions.
- **Logging:** Catching an exception during execution provides an opportunity to log critical information that could be used to detect, reproduce, and fix issues reliably. Log the exception, not just the message, but the body, inner exception and other useful artifacts that will be helpful later.
- **Do not catch and ignore exceptions:** One of the worst things you can do is catch an exception and do nothing with it. If you catch a generic exception, log it somewhere. If you don't log errors, it's difficult to investigate bugs and reported issues.

Retries

Implementing retry logic in an event streaming architecture can be complex. Supporting cancellation tokens, retry counts and exponential back off strategies are just a few of the considerations that make it challenging. Fortunately, Functions provides retry policies that can make up for many of these tasks that you would typically code yourself.

Several important factors that must be considered when using the retry policies with the Event Hub binding, include:

- **Avoid indefinite retries:** When the max retry count setting is set to a value of -1, the function will retry indefinitely. In general, indefinite retries should be used sparingly with Functions and almost never with the Event Hub trigger binding.
- **Choose the appropriate retry strategy:** A fixed delay strategy may be optimal for scenarios that receive back pressure from other Azure services. In these cases, the delay can help avoid throttling and other limitations encountered from those services. The exponential back off strategy offers more flexibility for retry delay intervals and is commonly used when integrating with third-party services, REST endpoints, and other Azure services.
- **Keep intervals and retry counts low:** When possible, try to maintain a retry interval shorter than one minute. Also, keep the maximum number of retry attempts to a reasonably low number. These settings are especially pertinent when running in the Functions Consumption plan.
- **Circuit breaker pattern:** A transient fault error from time to time is expected and a natural use case for retries. However, if a significant number of failures or issues are occurring during the processing of the function, it may make sense to stop the function, address the issues and restart later.

An important takeaway for the retry policies in Functions is that it is a best effort feature for reprocessing events. It does not substitute the need for error handling, logging, and other important patterns that provide resiliency to your code.

Strategies for failures and corrupt data

There are several noteworthy approaches that you can use to compensate for issues that arise due to failures or bad data in an event stream. Some fundamental strategies are:

- **Stop sending and reading:** Pause the reading and writing of events to fix the underlying issue. The

benefit of this approach is that data won't be lost, and operations can resume after a fix is rolled out. This approach may require a circuit-breaker component in the architecture and possibly a notification to the affected services to achieve a pause. In some cases, stopping a function may be necessary until the issues are resolved.

- **Drop messages:** If messages aren't important or are considered non-mission critical, consider moving on and not processing them. This doesn't work for scenarios that require strong consistency such as recording moves in a chess match or finance-based transactions. Error handling inside of a function is recommended for catching and dropping messages that can't be processed.
- **Retry:** There are many situations that may warrant the reprocessing of an event. The most common scenario would be a transient error encountered when calling another service or dependency. Network errors, service limits and availability, and strong consistency are perhaps the most frequent use cases that justify reprocessing attempts.
- **Dead letter:** The idea here is to publish the event to a different event hub so that the existing flow is not interrupted. The perception is that it has been moved off the hot path and could be dealt with later or by a different process. This solution is used frequently for handling poisoned messages or events. It should be noted that each function, that is configured with a different consumer group, will still encounter the bad or corrupt data in their stream and must handle it responsibly.
- **Retry and dead letter:** The combination of numerous retry attempts before ultimately publishing to a dead letter stream once a threshold is met, is another familiar method.
- **Use a schema registry:** A schema registry can be used as a proactive tool to help improve consistency and data quality. The Azure Schema Registry can support the transition of schemas along with versioning and different compatibility modes as schemas evolve. At its core, the schema will serve as a contract between producers and consumers, which could reduce the possibility of invalid or corrupt data being published to the stream.

In the end, there isn't a perfect solution and the consequences and tradeoffs of each of the strategies needs to be thoroughly examined. Based on the requirements, using several of these techniques together may be the best approach.

Next steps

Before continuing, consider reviewing these related articles:

- Azure Functions reliable event processing
- Designing Azure Functions for identical input
- Azure Functions error handling and retry guidance

Security

Related resources

- Monitoring serverless event processing provides guidance on monitoring serverless event-driven architectures.
- Serverless event processing is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- De-batching and filtering in serverless event processing with Event Hubs describes in more detail how these portions of the reference architecture work.

Secure Azure Functions with Event Hubs

11/19/2021 • 4 minutes to read • Edit Online

When configuring access to resources in Azure, you should apply fine-grained control over permissions to resources. Access to these resources should be based on *need to know* and *least privilege* security principles to make sure that clients can only perform the limited set of actions assigned to them.

Authorizing Access to Event Hubs

Authorizing access to Azure Event Hubs resources can be done using the following security constructs:

- **Azure Active Directory:** Azure Active Directory (Azure AD) provides role-based access control (RBAC) for granular control over a client's access to Event Hubs resources. Based on roles and permissions granted, Azure AD will authorize requests using an OAuth 2.0 access token.
- **Shared access signature:** A shared access signature (SAS) offers the ability to protect Event Hubs resources based on authorization rules. You define authorization policies by selecting one or more policy rules, such as the ability to send messages, listen to messages, and manage the entities in the namespace.

Shared access signature considerations

When using a shared access signature with Azure Functions and Event Hubs, the following considerations should be reviewed:

- **Avoid the Manage right:** In addition to being able to manage the entities in an Event Hubs namespace, the Manage right includes both Send and Listen rights. Ideally, a function app should only be granted a combination of the Send and Listen rights, based on the actions they perform.
- **Don't use the default Manage rule**: Avoid using the default policy rule named *RootManageSharedAccessKey* unless it's needed by your function app, which should be an uncommon scenario. Another caveat to this default rule is that it's created at the namespace level and grants permissions to all underlying event hubs.
- **Review shared access policy scopes:** Shared access policies can be created at the namespace level and per event hub. Consider creating granular access policies that are tailored for each client to limit their range and permissions.

Managed identity

An Active Directory identity can be assigned to a managed resource in Azure such as a function app or web app. Once an identity is assigned, it has the capabilities to work with other resources that use Azure AD for authorization, much like a service principal.

Function apps can be assigned a managed identity and take advantage of identity-based connections for a subset of services, including Event Hubs. Identity-based connections provide support for both the trigger and output binding extensions and must use the Event Hubs extension 5.x and higher for support.

Network

By default, Event Hubs namespaces are accessible from the internet, so long as the request comes with valid authentication and authorization. There are three options for limiting network access to Event Hubs namespaces:

- Allow access from specific IP addresses
- Allow access from specific virtual networks (service endpoints)

- Allow access via private endpoints

In all cases, it's important to note that at least one IP firewall rule or virtual network rule for the namespace is specified. Otherwise, if no IP address or virtual network rule is specified, the namespace is accessible over the public internet (using the access key).

Azure Functions can be configured to consume events from or publish events to event hubs, which are set up with either service endpoints or private endpoints. Regional virtual network integration is needed for your function app to connect to an event hub using a service endpoint or a private endpoint.

When setting up Functions to work with a private endpoint enabled resource, you need to set the `WEBSITE_VNET_ROUTE_ALL` application setting to `1`. If you want to fully lock down your function app, you also need to restrict your storage account.

To trigger (consume) events in a virtual network environment, the function app needs to be hosted in a Premium plan, a Dedicated (App Service) plan, or an App Service Environment (ASE).

Additionally, running in an Azure Functions Premium plan and consuming events from a virtual network restricted Event Hub requires virtual network trigger support, also referred to as runtime scale monitoring. Runtime scale monitoring can be configured via the Azure portal, Azure CLI, or other deployment solutions. Runtime scale monitoring isn't available when the function is running in a Dedicated (App Service) plan or an ASE.

To use runtime scale monitoring with Event Hubs, you need to use version 4.1.0 or higher of the Microsoft.Azure.WebJobs.Extensions.EventHubs extension.

Next steps

Before continuing, consider reviewing these related articles:

- Authorize access with Azure Active Directory
- Authorize access with a shared access signature in Azure Event Hubs
- Configure an identity-based resource

Observability

Related resources

- Monitoring serverless event processing provides guidance on monitoring serverless event-driven architectures.
- Serverless event processing is a reference architecture detailing a typical architecture of this type, with code samples and discussion of important considerations.
- De-batching and filtering in serverless event processing with Event Hubs describes in more detail how these portions of the reference architecture work.

DevOps Checklist

11/19/2021 • 14 minutes to read • Edit Online

DevOps is the integration of development, quality assurance, and IT operations into a unified culture and set of processes for delivering software. Use this checklist as a starting point to assess your DevOps culture and process.

Culture

Ensure business alignment across organizations and teams. Conflicts over resources, purpose, goals, and priorities within an organization can be a risk to successful operations. Ensure that the business, development, and operations teams are all aligned.

Ensure the entire team understands the software lifecycle. Your team needs to understand the overall lifecycle of the application, and which part of the lifecycle the application is currently in. This helps all team members know what they should be doing now, and what they should be planning and preparing for in the future.

Reduce cycle time. Aim to minimize the time it takes to move from ideas to usable developed software. Limit the size and scope of individual releases to keep the test burden low. Automate the build, test, configuration, and deployment processes whenever possible. Clear any obstacles to communication among developers, and between developers and operations.

Review and improve processes. Your processes and procedures, both automated and manual, are never final. Set up regular reviews of current workflows, procedures, and documentation, with a goal of continual improvement.

Do proactive planning. Proactively plan for failure. Have processes in place to quickly identify issues when they occur, escalate to the correct team members to fix, and confirm resolution.

Learn from failures. Failures are inevitable, but it's important to learn from failures to avoid repeating them. If an operational failure occurs, triage the issue, document the cause and solution, and share any lessons that were learned. Whenever possible, update your build processes to automatically detect that kind of failure in the future.

Optimize for speed and collect data. Every planned improvement is a hypothesis. Work in the smallest increments possible. Treat new ideas as experiments. Instrument the experiments so that you can collect production data to assess their effectiveness. Be prepared to fail fast if the hypothesis is wrong.

Allow time for learning. Both failures and successes provide good opportunities for learning. Before moving on to new projects, allow enough time to gather the important lessons, and make sure those lessons are absorbed by your team. Also give the team the time to build skills, experiment, and learn about new tools and techniques.

Document operations. Document all tools, processes, and automated tasks with the same level of quality as your product code. Document the current design and architecture of any systems you support, along with recovery processes and other maintenance procedures. Focus on the steps you actually perform, not theoretically optimal processes. Regularly review and update the documentation. For code, make sure that meaningful comments are included, especially in public APIs, and use tools to automatically generate code documentation whenever possible.

Share knowledge. Documentation is only useful if people know that it exists and can find it. Ensure the documentation is organized and easily discoverable. Be creative: Use brown bags (informal presentations),

videos, or newsletters to share knowledge.

Development

Provide developers with production-like environments. If development and test environments don't match the production environment, it is hard to test and diagnose problems. Therefore, keep development and test environments as close to the production environment as possible. Make sure that test data is consistent with the data used in production, even if it's sample data and not real production data (for privacy or compliance reasons). Plan to generate and anonymize sample test data.

Ensure that all authorized team members can provision infrastructure and deploy the application. Setting up production-like resources and deploying the application should not involve complicated manual tasks or detailed technical knowledge of the system. Anyone with the right permissions should be able to create or deploy production-like resources without going to the operations team.

> This recommendation doesn't imply that anyone can push live updates to the production deployment. It's about reducing friction for the development and QA teams to create production-like environments.

Instrument the application for insight. To understand the health of your application, you need to know how it's performing and whether it's experiencing any errors or problems. Always include instrumentation as a design requirement, and build the instrumentation into the application from the start. Instrumentation must include event logging for root cause analysis, but also telemetry and metrics to monitor the overall health and usage of the application.

Track your technical debt. In many projects, release schedules can get prioritized over code quality to one degree or another. Always keep track when this occurs. Document any shortcuts or other suboptimal implementations, and schedule time in the future to revisit these issues.

Consider pushing updates directly to production. To reduce the overall release cycle time, consider pushing properly tested code commits directly to production. Use feature toggles to control which features are enabled. This allows you to move from development to release quickly, using the toggles to enable or disable features. Toggles are also useful when performing tests such as canary releases, where a particular feature is deployed to a subset of the production environment.

Testing

Automate testing. Manually testing software is tedious and susceptible to error. Automate common testing tasks and integrate the tests into your build processes. Automated testing ensures consistent test coverage and reproducibility. Integrated UI tests should also be performed by an automated tool. Azure offers development and test resources that can help you configure and execute testing. For more information, see Development and test.

Test for failures. If a system can't connect to a service, how does it respond? Can it recover once the service is available again? Make fault injection testing a standard part of review on test and staging environments. When your test process and practices are mature, consider running these tests in production.

Test in production. The release process doesn't end with deployment to production. Have tests in place to ensure that deployed code works as expected. For deployments that are infrequently updated, schedule production testing as a regular part of maintenance.

Automate performance testing to identify performance issues early. The impact of a serious performance issue can be as severe as a bug in the code. While automated functional tests can prevent application bugs, they might not detect performance problems. Define acceptable performance goals for metrics like latency, load times, and resource usage. Include automated performance tests in your release pipeline, to make sure the application meets those goals.

Perform capacity testing. An application might work fine under test conditions, and then have problems in production due to scale or resource limitations. Always define the maximum expected capacity and usage limits. Test to make sure the application can handle those limits, but also test what happens when those limits are exceeded. Capacity testing should be performed at regular intervals.

After the initial release, you should run performance and capacity tests whenever updates are made to production code. Use historical data to fine-tune tests and to determine what types of tests need to be performed.

Perform automated security penetration testing. Ensuring your application is secure is as important as testing any other functionality. Make automated penetration testing a standard part of the build and deployment process. Schedule regular security tests and vulnerability scanning on deployed applications, monitoring for open ports, endpoints, and attacks. Automated testing does not remove the need for in-depth security reviews at regular intervals.

Perform automated business continuity testing. Develop tests for large-scale business continuity, including backup recovery and failover. Set up automated processes to perform these tests regularly.

Release

Automate deployments. Automate deploying the application to test, staging, and production environments. Automation enables faster and more reliable deployments, and ensures consistent deployments to any supported environment. It removes the risk of human error caused by manual deployments. It also makes it easy to schedule releases for convenient times, to minimize any effects of potential downtime. Have systems in place to detect any problems during rollout, and have an automated way to roll forward fixes or roll back changes.

Use continuous integration. Continuous integration (CI) is the practice of merging all developer code into a central codebase on a regular schedule, and then automatically performing standard build and test processes. CI ensures that an entire team can work on a codebase at the same time without having conflicts. It also ensures that code defects are found as early as possible. Preferably, the CI process should run every time that code is committed or checked in. At the very least, it should run once per day.

> Consider adopting a trunk based development model. In this model, developers commit to a single branch (the trunk). There is a requirement that commits never break the build. This model facilitates CI, because all feature work is done in the trunk, and any merge conflicts are resolved when the commit happens.

Consider using continuous delivery. Continuous delivery (CD) is the practice of ensuring that code is always ready to deploy, by automatically building, testing, and deploying code to production-like environments. Adding continuous delivery to create a full CI/CD pipeline will help you detect code defects as soon as possible, and ensures that properly tested updates can be released in a very short time.

> Continuous *deployment* is an additional process that automatically takes any updates that have passed through the CI/CD pipeline and deploys them into production. Continuous deployment requires robust automatic testing and advanced process planning, and may not be appropriate for all teams.

Make small incremental changes. Large code changes have a greater potential to introduce bugs. Whenever possible, keep changes small. This limits the potential effects of each change, and makes it easier to understand and debug any issues.

Control exposure to changes. Make sure you're in control of when updates are visible to your end users. Consider using feature toggles to control when features are enabled for end users.

Implement release management strategies to reduce deployment risk. Deploying an application update to production always entails some risk. To minimize this risk, use strategies such as canary releases or

blue-green deployments to deploy updates to a subset of users. Confirm the update works as expected, and then roll the update out to the rest of the system.

Document all changes. Minor updates and configuration changes can be a source of confusion and versioning conflict. Always keep a clear record of any changes, no matter how small. Log everything that changes, including patches applied, policy changes, and configuration changes. (Don't include sensitive data in these logs. For example, log that a credential was updated, and who made the change, but don't record the updated credentials.) The record of the changes should be visible to the entire team.

Consider making infrastructure immutable. Immutable infrastructure is the principle that you shouldn't modify infrastructure after it's deployed to production. Otherwise, you can get into a state where ad hoc changes have been applied, making it hard to know exactly what changed. Immutable infrastructure works by replacing entire servers as part of any new deployment. This allows the code and the hosting environment to be tested and deployed as a block. Once deployed, infrastructure components aren't modified until the next build and deploy cycle.

Monitoring

Make systems observable. The operations team should always have clear visibility into the health and status of a system or service. Set up external health endpoints to monitor status, and ensure that applications are coded to instrument the operations metrics. Use a common and consistent schema that helps you correlate events across systems. Azure Diagnostics and Application Insights are the standard method of tracking the health and status of Azure resources. Azure Monitor also provides centralized monitoring and management for cloud or hybrid solutions.

Aggregate and correlate logs and metrics. A properly instrumented telemetry system will provide a large amount of raw performance data and event logs. Make sure that telemetry and log data is processed and correlated in a short period of time, so that operations staff always have an up-to-date picture of system health. Organize and display data in ways that give a cohesive view of any issues, so that whenever possible it's clear when events are related to one another.

> Consult your corporate retention policy for requirements on how data is processed and how long it should be stored.

Implement automated alerts and notifications. Set up monitoring tools like Azure Monitor to detect patterns or conditions that indicate potential or current issues, and send alerts to the team members who can address the issues. Tune the alerts to avoid false positives.

Monitor assets and resources for expirations. Some resources and assets, such as certificates, expire after a given amount of time. Make sure to track which assets expire, when they expire, and what services or features depend on them. Use automated processes to monitor these assets. Notify the operations team before an asset expires, and escalate if expiration threatens to disrupt the application.

Management

Automate operations tasks. Manually handling repetitive operations processes is error-prone. Automate these tasks whenever possible to ensure consistent execution and quality. Code that implements the automation should be versioned in source control. As with any other code, automation tools must be tested.

Take an infrastructure-as-code approach to provisioning. Minimize the amount of manual configuration needed to provision resources. Instead, use scripts and Azure Resource Manager templates. Keep the scripts and templates in source control, like any other code you maintain.

Consider using containers. Containers provide a standard package-based interface for deploying applications. Using containers, an application is deployed using self-contained packages that include any

software, dependencies, and files needed to run the application, which greatly simplifies the deployment process.

Containers also create an abstraction layer between the application and the underlying operating system, which provides consistency across environments. This abstraction can also isolate a container from other processes or applications running on a host.

Implement resiliency and self-healing. Resiliency is the ability of an application to recover from failures. Strategies for resiliency include retrying transient failures, and failing over to a secondary instance or even another region. For more information, see Designing reliable Azure applications . Instrument your applications so that issues are reported immediately and you can manage outages or other system failures.

Have an operations manual. An operations manual or *runbook* documents the procedures and management information needed for operations staff to maintain a system. Also document any operations scenarios and mitigation plans that might come into play during a failure or other disruption to your service. Create this documentation during the development process, and keep it up to date afterwards. This is a living document, and should be reviewed, tested, and improved regularly.

Shared documentation is critical. Encourage team members to contribute and share knowledge. The entire team should have access to documents. Make it easy for anyone on the team to help keep documents updated.

Document on-call procedures. Make sure on-call duties, schedules, and procedures are documented and shared to all team members. Keep this information up-to-date at all times.

Document escalation procedures for third-party dependencies. If your application depends on external third-party services that you don't directly control, you must have a plan to deal with outages. Create documentation for your planned mitigation processes. Include support contacts and escalation paths.

Use configuration management. Configuration changes should be planned, visible to operations, and recorded. This could take the form of a configuration management database, or a configuration-as-code approach. Configuration should be audited regularly to ensure that what's expected is actually in place.

Get an Azure support plan and understand the process. Azure offers a number of support plans. Determine the right plan for your needs, and make sure the entire team knows how to use it. Team members should understand the details of the plan, how the support process works, and how to open a support ticket with Azure. If you are anticipating a high-scale event, Azure support can assist you with increasing your service limits. For more information, see the Azure Support FAQs.

Follow least-privilege principles when granting access to resources. Carefully manage access to resources. Access should be denied by default, unless a user is explicitly given access to a resource. Only grant a user access to what they need to complete their tasks. Track user permissions and perform regular security audits.

Use Azure role-based access control. Assigning user accounts and access to resources should not be a manual process. Use Azure role-based access control (Azure RBAC) grant access based on Azure Active Directory identities and groups.

Use a bug tracking system to track issues. Without a good way to track issues, it's easy to miss items, duplicate work, or introduce additional problems. Don't rely on informal person-to-person communication to track the status of bugs. Use a bug tracking tool to record details about problems, assign resources to address them, and provide an audit trail of progress and status.

Manage all resources in a change management system. All aspects of your DevOps process should be included in a management and versioning system, so that changes can be easily tracked and audited. This includes code, infrastructure, configuration, documentation, and scripts. Treat all these types of resources as code throughout the test/build/review process.

Use checklists. Create operations checklists to ensure processes are followed. It's common to miss something

in a large manual, and following a checklist can force attention to details that might otherwise be overlooked. Maintain the checklists, and continually look for ways to automate tasks and streamline processes.

For more about DevOps, see What is DevOps? on the Visual Studio site.

Advanced Azure Resource Manager template functionality

11/19/2021 • 2 minutes to read • Edit Online

This section provides advanced examples for Azure Resource Manager templates.

Update a resource. You may need to update a resource during a deployment. You might encounter this scenario when you cannot specify all the properties for a resource until other, dependent resources are created.

Use an object parameter in a copy loop. There is a limit of 256 parameters per deployment. Once you get to larger and more complex deployments you may run out of parameters. One way to solve this problem is to use an object as a parameter instead of a value.

Property transformer and collector. A property transform and collector template can transform objects into the JSON schema expected by a nested template.

> **NOTE**
>
> These articles assume you have an advanced understanding of Azure Resource Manager templates.

Update a resource in an Azure Resource Manager template

11/19/2021 • 3 minutes to read • Edit Online

There are some scenarios in which you need to update a resource during a deployment. You might encounter this scenario when you cannot specify all the properties for a resource until other, dependent resources are created. For example, if you create a backend pool for a load balancer, you might update the network interfaces (NICs) on your virtual machines (VMs) to include them in the backend pool. And while Resource Manager supports updating resources during deployment, you must design your template correctly to avoid errors and to ensure the deployment is handled as an update.

First, you must reference the resource once in the template to create it and then reference the resource by the same name to update it later. However, if two resources have the same name in a template, Resource Manager throws an exception. To avoid this error, specify the updated resource in a second template that's either linked or included as a subtemplate using the `Microsoft.Resources/deployments` resource type.

Second, you must either specify the name of the existing property to change or a new name for a property to add in the nested template. You must also specify the original properties and their original values. If you fail to provide the original properties and values, Resource Manager assumes you want to create a new resource and deletes the original resource.

Example template

Let's look at an example template that demonstrates this. Our template deploys a virtual network named `firstVNet` that has one subnet named `firstSubnet`. It then deploys a virtual network interface (NIC) named `nic1` and associates it with our subnet. Then, a deployment resource named `updateVNet` includes a nested template that updates our `firstVNet` resource by adding a second subnet named `secondSubnet`.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {},
    "resources": [
        {
            "apiVersion": "2020-05-01",
            "name": "firstVNet",
            "location": "[resourceGroup().location]",
            "type": "Microsoft.Network/virtualNetworks",
            "properties": {
                "addressSpace": {
                    "addressPrefixes": [
                        "10.0.0.0/22"
                    ]
                },
                "subnets": [
                    {
                        "name": "firstSubnet",
                        "properties": {
                            "addressPrefix": "10.0.0.0/24"
                        }
                    }
                ]
            }
        },
        {
```

```
            "apiVersion": "2020-05-01",
            "type": "Microsoft.Network/networkInterfaces",
            "name": "nic1",
            "location": "[resourceGroup().location]",
            "dependsOn": [
                "firstVNet"
            ],
            "properties": {
                "ipConfigurations": [
                    {
                        "name": "ipconfig1",
                        "properties": {
                            "privateIPAllocationMethod": "Dynamic",
                            "subnet": {
                                "id": "[resourceId('Microsoft.Network/virtualNetworks/subnets', 'firstVNet',
'firstSubnet')]"
                            }
                        }
                    }
                ]
            }
        },
        {
            "apiVersion": "2020-06-01",
            "type": "Microsoft.Resources/deployments",
            "name": "updateVNet",
            "dependsOn": [
                "nic1"
            ],
            "properties": {
                "mode": "Incremental",
                "parameters": {},
                "template": {
                    "$schema": "https://schema.management.azure.com/schemas/2019-04-
01/deploymentTemplate.json#",
                    "contentVersion": "1.0.0.1",
                    "parameters": {},
                    "variables": {},
                    "resources": [
                        {
                            "apiVersion": "2020-05-01",
                            "name": "firstVNet",
                            "location": "[resourceGroup().location]",
                            "type": "Microsoft.Network/virtualNetworks",
                            "properties": {
                                "addressSpace": "[reference('firstVNet').addressSpace]",
                                "subnets": [
                                    {
                                        "name": "[reference('firstVNet').subnets[0].name]",
                                        "properties": {
                                            "addressPrefix": "
[reference('firstVNet').subnets[0].properties.addressPrefix]"
                                        }
                                    },
                                    {
                                        "name": "secondSubnet",
                                        "properties": {
                                            "addressPrefix": "10.0.1.0/24"
                                        }
                                    }
                                ]
                            }
                        }
                    ],
                    "outputs": {}
                }
            }
        }
    ],
```

```
    "outputs": {}
}
```

Let's take a look at the resource object for our `firstVNet` resource first. Notice that we specify again the settings for our `firstVNet` in a nested template—this is because Resource Manager doesn't allow the same deployment name within the same template and nested templates are considered to be a different template. By again specifying our values for our `firstSubnet` resource, we are telling Resource Manager to update the existing resource instead of deleting it and redeploying it. Finally, our new settings for `secondSubnet` are picked up during this update.

Try the template

An example template is available on GitHub. To deploy the template, run the following Azure CLI commands:

```
az group create --location <location> --name <resource-group-name>
az deployment group create -g <resource-group-name> \
    --template-uri https://raw.githubusercontent.com/mspnp/template-examples/master/example1-
update/deploy.json
```

Once deployment has finished, open the resource group you specified in the portal. You see a virtual network named `firstVNet` and a NIC named `nic1`. Click `firstVNet`, then click `subnets`. You see the `firstSubnet` that was originally created, and you see the `secondSubnet` that was added in the `updateVNet` resource.

NAME	ADDRESS RANGE	AVAILABLE ADDRESSES	SECURITY GROUP	
firstSubnet	10.0.0.0/24	250	-	...
secondSubnet	10.0.1.0/24	251	-	...

Then, go back to the resource group and click `nic1` then click `IP configurations`. In the `IP configurations` section, the `subnet` is set to `firstSubnet (10.0.0.0/24)`.

IP configurations

* Subnet firstSubnet (10.0.0.0/24)

Search IP configurations

NAME	IP VERSION	TYPE	PRIVATE IP ADDRESS	PUBLIC IP ADDRESS	
ipconfig1	IPv4	Primary	10.0.0.4 (Dynamic)	-	...

The original `firstVNet` has been updated instead of re-created. If `firstVNet` had been re-created, `nic1` would not be associated with `firstVNet`.

Next steps

- Learn how to Use an object as a parameter in an Azure Resource Manager template.

Use objects as parameters in a copy loop in an Azure Resource Manager template

11/19/2021 • 2 minutes to read • Edit Online

When using objects as a parameter in Azure Resource Manager templates you may want to include them in a copy loop, so here is an example that uses them in that way:

This approach becomes very useful when combined with the serial copy loop, particularly for deploying child resources.

To demonstrate this, let's look at a template that deploys a network security group (NSG) with two security rules.

First, let's take a look at our parameters. When we look at our template we'll see that we've defined one parameter named `networkSecurityGroupsSettings` that includes an array named `securityRules`. This array contains two JSON objects that specify a number of settings for a security rule.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters":{
        "networkSecurityGroupsSettings": {
            "value": {
                "securityRules": [
                    {
                        "name": "RDPAllow",
                        "description": "allow RDP connections",
                        "direction": "Inbound",
                        "priority": 100,
                        "sourceAddressPrefix": "*",
                        "destinationAddressPrefix": "10.0.0.0/24",
                        "sourcePortRange": "*",
                        "destinationPortRange": "3389",
                        "access": "Allow",
                        "protocol": "Tcp"
                    },
                    {
                        "name": "HTTPAllow",
                        "description": "allow HTTP connections",
                        "direction": "Inbound",
                        "priority": 200,
                        "sourceAddressPrefix": "*",
                        "destinationAddressPrefix": "10.0.1.0/24",
                        "sourcePortRange": "*",
                        "destinationPortRange": "80",
                        "access": "Allow",
                        "protocol": "Tcp"
                    }
                ]
            }
        }
    }
}
```

Now let's take a look at our template. We have a resource named `NSG1` deploys the NSG, it also leverages ARM's built-in property iteration feature; by adding copy loop to the properties section of a resource in your template, you can dynamically set the number of items for a property during deployment. You also avoid having to repeat template syntax.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "VNetSettings": {
            "type": "object"
        },
        "networkSecurityGroupsSettings": {
            "type": "object"
        }
    },
    "resources": [
        {
            "apiVersion": "2020-05-01",
            "type": "Microsoft.Network/virtualNetworks",
            "name": "[parameters('VNetSettings').name]",
            "location": "[resourceGroup().location]",
            "properties": {
                "addressSpace": {
                    "addressPrefixes": [
                        "[parameters('VNetSettings').addressPrefixes[0].addressPrefix]"
                    ]
                },
                "subnets": [
                    {
                        "name": "[parameters('VNetSettings').subnets[0].name]",
                        "properties": {
                            "addressPrefix": "[parameters('VNetSettings').subnets[0].addressPrefix]"
                        }
                    },
                    {
                        "name": "[parameters('VNetSettings').subnets[1].name]",
                        "properties": {
                            "addressPrefix": "[parameters('VNetSettings').subnets[1].addressPrefix]"
                        }
                    }
                ]
            }
        },
        {
            "apiVersion": "2020-05-01",
            "type": "Microsoft.Network/networkSecurityGroups",
            "name": "NSG1",
            "location": "[resourceGroup().location]",
            "properties": {
                "copy": [
                    {
                        "name": "securityRules",
                        "count": "[length(parameters('networkSecurityGroupsSettings').securityRules)]",
                        "input": {
                            "description": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].description]",
                            "priority": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].priority]",
                            "protocol": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].protocol]",
                            "sourcePortRange": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].sourcePortRange]",
                            "destinationPortRange": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].destinationPortRange]",
                            "sourceAddressPrefix": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].sourceAddressPrefix]",
                            "destinationAddressPrefix": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].destinationAddressPrefix]",
                            "access": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].access]",
                            "direction": "
[parameters('networkSecurityGroupsSettings').securityRules[copyIndex()].direction]"
```

```
                        }
                    }
                ]
            }
        }
    ]
}
```

Let's take a closer look at how we specify our property values in the `securityRules` child resource. All of our properties are referenced using the `parameters()` function, and then we use the dot operator to reference our `securityRules` array, indexed by the current value of the iteration. Finally, we use another dot operator to reference the name of the object.

Try the template

An example template is available on GitHub. To deploy the template, clone the repo and run the following Azure CLI commands:

```
git clone https://github.com/mspnp/template-examples.git
cd template-examples/example3-object-param
az group create --location <location> --name <resource-group-name>
az deployment group create -g <resource-group-name> \
    --template-uri https://raw.githubusercontent.com/mspnp/template-examples/master/example3-object-
param/deploy.json \
    --parameters deploy.parameters.json
```

Next steps

- Learn how to create a template that iterates through an object array and transforms it into a JSON schema. See Implement a property transformer and collector in an Azure Resource Manager template

Implement a property transformer and collector in an Azure Resource Manager template

11/19/2021 • 6 minutes to read • Edit Online

In use an object as a parameter in an Azure Resource Manager template, you learned how to store resource property values in an object and apply them to a resource during deployment. While this is a very useful way to manage your parameters, it still requires you to map the object's properties to resource properties each time you use it in your template.

To work around this, you can implement a property transform and collector template that iterates your object array and transforms it into the JSON schema expected by the resource.

> **IMPORTANT**
>
> This approach requires that you have a deep understanding of Resource Manager templates and functions.

Let's take a look at how we can implement a property collector and transformer with an example that deploys a network security group. The diagram below shows the relationship between our templates and our resources within those templates:

Our **calling template** includes two resources:

- A template link that invokes our **collector template**.
- The network security group resource to deploy.

Our **collector template** includes two resources:

- An **anchor** resource.
- A template link that invokes the transform template in a copy loop.

Our **transform template** includes a single resource: an empty template with a variable that transforms our `source` JSON to the JSON schema expected by our network security group resource in the **main template**.

Parameter object

We'll be using our `securityRules` parameter object from objects as parameters. Our **transform template** will transform each object in the `securityRules` array into the JSON schema expected by the network security group resource in our **calling template**.

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "networkSecurityGroupsSettings": {
            "value": {
                "securityRules": [
                    {
                        "name": "RDPAllow",
                        "description": "allow RDP connections",
                        "direction": "Inbound",
                        "priority": 100,
                        "sourceAddressPrefix": "*",
                        "destinationAddressPrefix": "10.0.0.0/24",
                        "sourcePortRange": "*",
                        "destinationPortRange": "3389",
                        "access": "Allow",
                        "protocol": "Tcp"
                    },
                    {
                        "name": "HTTPAllow",
                        "description": "allow HTTP connections",
                        "direction": "Inbound",
                        "priority": 200,
                        "sourceAddressPrefix": "*",
                        "destinationAddressPrefix": "10.0.1.0/24",
                        "sourcePortRange": "*",
                        "destinationPortRange": "80",
                        "access": "Allow",
                        "protocol": "Tcp"
                    }
                ]
            }
        }
    }
}
```

Let's look at our **transform template** first.

Transform template

Our **transform template** includes two parameters that are passed from the **collector template**:

- `source` is an object that receives one of the property value objects from the property array. In our example, each object from the `"securityRules"` array will be passed in one at a time.
- `state` is an array that receives the concatenated results of all the previous transforms. This is the collection of transformed JSON.

Our parameters look like this:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "source": {
            "type": "object"
        },
        "state": {
            "type": "array",
            "defaultValue": []
        }
    },
```

Our template also defines a variable named `instance`. It performs the actual transform of our `source` object into the required JSON schema:

```
"variables": {
    "instance": [
        {
            "name": "[parameters('source').name]",
            "properties": {
                "description": "[parameters('source').description]",
                "protocol": "[parameters('source').protocol]",
                "sourcePortRange": "[parameters('source').sourcePortRange]",
                "destinationPortRange": "[parameters('source').destinationPortRange]",
                "sourceAddressPrefix": "[parameters('source').sourceAddressPrefix]",
                "destinationAddressPrefix": "[parameters('source').destinationAddressPrefix]",
                "access": "[parameters('source').access]",
                "priority": "[parameters('source').priority]",
                "direction": "[parameters('source').direction]"
            }
        }
    ]
}
```

Finally, the `output` of our template concatenates the collected transforms of our `state` parameter with the current transform performed by our `instance` variable:

```
"resources": [],
"outputs": {
    "collection": {
        "type": "array",
        "value": "[concat(parameters('state'), variables('instance'))]"
    }
}
```

Next, let's take a look at our **collector template** to see how it passes in our parameter values.

Collector template

Our **collector template** includes three parameters:

- `source` is our complete parameter object array. It's passed in by the **calling template**. This has the same name as the `source` parameter in our **transform template** but there is one key difference that you may have already noticed: this is the complete array, but we only pass one element of this array to the **transform template** at a time.
- `transformTemplateUri` is the URI of our **transform template**. We're defining it as a parameter here for template reusability.
- `state` is an initially empty array that we pass to our **transform template**. It stores the collection of transformed parameter objects when the copy loop is complete.

Our parameters look like this:

```
"parameters": {
    "source": {
        "type": "array"
    },
    "transformTemplateUri": {
        "type": "string"
    },
    "state": {
        "type": "array",
        "defaultValue": []
    }
}
```

Next, we define a variable named `count`. Its value is the length of the `source` parameter object array:

```
"variables": {
    "count": "[length(parameters('source'))]"
}
```

As you might suspect, we use it for the number of iterations in our copy loop.

Now let's take a look at our resources. We define two resources:

- `loop-0` is the zero-based resource for our copy loop.
- `loop-` is concatenated with the result of the `copyIndex(1)` function to generate a unique iteration-based name for our resource, starting with `1`.

Our resources look like this:

```
"resources": [
    {
        "type": "Microsoft.Resources/deployments",
        "apiVersion": "2015-01-01",
        "name": "loop-0",
        "properties": {
            "mode": "Incremental",
            "parameters": { },
            "template": {
                "$schema": "https://schema.management.azure.com/schemas/2015-01-
01/deploymentTemplate.json#",
                "contentVersion": "1.0.0.0",
                "parameters": { },
                "variables": { },
                "resources": [ ],
                "outputs": {
                    "collection": {
                        "type": "array",
                        "value": "[parameters('state')]"
                    }
                }
            }
        }
    },
    {
        "type": "Microsoft.Resources/deployments",
        "apiVersion": "2015-01-01",
        "name": "[concat('loop-', copyindex(1))]",
        "copy": {
            "name": "iterator",
            "count": "[variables('count')]",
            "mode": "serial"
        },
        "dependsOn": [
            "loop-0"
        ],
        "properties": {
            "mode": "Incremental",
            "templateLink": { "uri": "[parameters('transformTemplateUri')]" },
            "parameters": {
                "source": { "value": "[parameters('source')[copyindex()]]" },
                "state": { "value": "[reference(concat('loop-', copyindex())).outputs.collection.value]" }
            }
        }
    }
]
```

Let's take a closer look at the parameters we're passing to our **transform template** in the nested template. Recall from earlier that our `source` parameter passes the current object in the `source` parameter object array. The `state` parameter is where the collection happens, because it takes the output of the previous iteration of our copy loop—notice that the `reference()` function uses the `copyIndex()` function with no parameter to reference the `name` of our previous linked template object—and passes it to the current iteration.

Finally, the `output` of our template returns the `output` of the last iteration of our **transform template**:

```
"outputs": {
    "result": {
        "type": "array",
        "value": "[reference(concat('loop-', variables('count'))).outputs.collection.value]"
    }
}
```

It may seem counterintuitive to return the `output` of the last iteration of our **transform template** to our

calling template because it appeared we were storing it in our `source` parameter. However, remember that it's the last iteration of our **transform template** that holds the complete array of transformed property objects, and that's what we want to return.

Finally, let's take a look at how to call the **collector template** from our **calling template**.

Calling template

Our **calling template** defines a single parameter named `networkSecurityGroupsSettings`:

```
...
"parameters": {
    "networkSecurityGroupsSettings": {
        "type": "object"
    }
}
```

Next, our template defines a single variable named `collectorTemplateUri`:

```
"variables": {
    "collectorTemplateUri": "[uri(deployment().properties.templateLink.uri, 'collector.template.json')]"
}
```

As you would expect, this is the URI for the **collector template** that will be used by our linked template resource:

```
{
    "apiVersion": "2020-06-01",
    "name": "collector",
    "type": "Microsoft.Resources/deployments",
    "properties": {
        "mode": "Incremental",
        "templateLink": {
            "uri": "[variables('collectorTemplateUri')]",
            "contentVersion": "1.0.0.0"
        },
        "parameters": {
            "source": {
                "value": "[parameters('networkSecurityGroupsSettings').securityRules]"
            },
            "transformTemplateUri": {
                "value": "[uri(deployment().properties.templateLink.uri, 'transform.json')]"
            }
        }
    }
}
```

We pass two parameters to the **collector template**:

- `source` is our property object array. In our example, it's our `networkSecurityGroupsSettings` parameter.
- `transformTemplateUri` is the variable we just defined with the URI of our **collector template**.

Finally, our `Microsoft.Network/networkSecurityGroups` resource directly assigns the `output` of the `collector` linked template resource to its `securityRules` property:

```
"resources": [
    {
        "apiVersion": "2020-05-01",
        "type": "Microsoft.Network/networkSecurityGroups",
        "name": "networkSecurityGroup1",
        "location": "[resourceGroup().location]",
        "properties": {
            "securityRules": "[reference('collector').outputs.result.value]"
        }
    }
],
"outputs": {
    "instance": {
        "type": "array",
        "value": "[reference('collector').outputs.result.value]"
    }
}
```

Try the template

An example template is available on GitHub. To deploy the template, clone the repo and run the following Azure CLI commands:

```
git clone https://github.com/mspnp/template-examples.git
cd template-examples/example4-collector
az group create --location <location> --name <resource-group-name>
az deployment group create -g <resource-group-name> \
    --template-uri https://raw.githubusercontent.com/mspnp/template-examples/master/example4-
collector/deploy.json \
    --parameters deploy.parameters.json
```

Build solutions for high availability using Availability Zones

11/19/2021 • 7 minutes to read • Edit Online

Microsoft Azure global infrastructure is designed and constructed at every layer to deliver the highest levels of redundancy and resiliency to its customers. Azure infrastructure is composed of geographies, regions, and Availability Zones, which limit the blast radius of a failure and therefore limit potential impact to customer applications and data. The Azure Availability Zones construct was developed to provide a software and networking solution to protect against datacenter failures and to provide increased high availability (HA) to our customers.

Availability Zones are unique physical locations within an Azure region. Each zone is made up of one or more datacenters with independent power, cooling, and networking. The physical separation of Availability Zones within a region limits the impact to applications and data from zone failures, such as large-scale flooding, major storms and superstorms, and other events that could disrupt site access, safe passage, extended utilities uptime, and the availability of resources. Availability Zones and their associated datacenters are designed such that if one zone is compromised, the services, capacity, and availability are supported by the other Availability Zones in the region.

Availability Zones can be used to spread a solution across multiple zones within a region, allowing for an application to continue functioning when one zone fails. With Availability Zones, Azure offers industry best 99.99% Virtual Machine (VM) uptime service-level agreement (SLA). Zone-redundant services replicate your services and data across Availability Zones to protect from single points of failure.

For additional information on Availability Zones, including service support by region and pricing, refer to What are Availability Zones in Azure? in Microsoft Azure Documentation.

Availability Zones reference architectures

The following architectures feature high-availability scenarios:

- High availability enterprise deployment using App Services Environment
- IaaS: Web application with relational database
- Multi-region load balancing with Traffic Manager and Application Gateway
- Multi-region web app with private connectivity to database
- Multi-tier web application built for HA/DR

Delivering reliability in Azure

Designing solutions that continue to function in spite of failure is key to improving the reliability of a solution. In cloud-based solutions, building to survive failure is a shared responsibility. This can be viewed at three levels: a resilient foundation, resilient services, and resilient applications. The foundation is the Microsoft investment in the platform, including Availability Zones. On top of this foundation are the Azure services that customers can enable to support high availability, such as zone-redundant storage (ZRS), which replicates data across zones. The customer builds applications upon the enabled services supported by the foundation. The applications should be architected to support resiliency.

Your applications

Your **app** or **workload** architecture

Resilient services

Azure capabilities you **enable as needed**

Resilient foundation

Azure capabilities **built into the platform**

When architecting for resilience, all three layers-foundation, services, and applications-should be considered to achieve the highest level of reliability. Since a solution can be made up of many components, each component should be designed for reliability.

Zonal vs. zone-redundant architecture

An Availability Zone in an Azure region is a combination of a fault domain and an update domain. For example, if you create three or more VMs across three zones in an Azure region, your VMs are effectively distributed across three fault domains and three update domains. The Azure platform recognizes this distribution across update domains to take care that VMs in different zones are not updated at the same time.

Azure services supporting Availability Zones fall into two categories: zonal and zone redundant. Customer workloads can be categorized to utilize either architecture scenario to meet application performance and durability requirements.

With *zonal* architecture, a resource can be deployed to a specific, self-selected Availability Zone to achieve more stringent latency or performance requirements. Resiliency is self-architected by replicating applications and data to one or more zones within the region. You can choose specific Availability Zones for synchronous replication, providing high availability, or asynchronous replication, providing backup or cost advantage. You can pin resources-for example, virtual machines, managed disks, or standard IP addresses-to a specific zone, allowing for increased resilience by having one or more instances of resources spread across zones.

With *zone-redundant* architecture, the Azure platform automatically replicates the resource and data across zones. Microsoft manages the delivery of high availability, since Azure automatically replicates and distributes instances within the region.

A failure to a zone affects zonal and zone-redundant services differently. In the case of a zone failure, the zonal services in the failed zone become unavailable until the zone has recovered. By architecting your solutions to use replicated VMs in zones, you can protect your applications and data from a zone becoming unavailable-for

example, due to a power outage. If one zone is compromised, replicated apps and data are instantly available in another zone.

Zonal architecture applies to a specific resource, typically an infrastructure as a service (IaaS) resource, like a VM or managed disk, as illustrated.

For example, zonal load balancer, VM, managed disks, virtual machine scale sets.

In the illustration, each VM and load balancer (LB) are deployed to a specific zone.

With zone-redundant services, the distribution of the workload is a feature of the service and is handled by Azure. Azure automatically replicates the resource across zones without requiring your intervention. ZRS, for example, replicates the data across three zones so a zone failure does not impact the HA of the data.

The following illustration is of a zone-redundant load balancer.

For example, zone-redundant load balancer, Azure Application Gateway, Azure Service Bus, virtual private network (VPN), zone-redundant storage, Azure ExpressRoute, Azure Event Hubs, Azure Cosmos DB.

A few resources, like the load balancer and subnets, support both zonal and zone-redundant deployments. An important consideration in HA is distributing the traffic effectively across resources in the different Availability Zones. For information on how Availability Zones apply to the load balancer resources for both zonal and zone-redundant resources, refer to Standard Load Balancer and Availability Zones.

The following is a summary of the zonal (Z) and zone-redundant (ZR) Azure services.

Analytics • Azure Data Explorer (ZR) • Azure Event Hubs (ZR) Compute • Linux virtual machines (Z) • Windows virtual machines (Z) • Virtual machine scale sets (Z, ZR) • Azure App Service Environments (Z) Containers • Azure Kubernetes Service (AKS) (Z) • Azure Service Fabric (Z) Databases • Azure SQL Database (ZR) • Azure Cache for Redis (Z, ZR) • Azure Cosmos DB (ZR) DevOps • Azure DevOps (ZR) Identity • Azure Active Directory Domain Services (ZR) Integration • Azure Event Grid (ZR) • Azure Service Bus (ZR)	Management and governance • Azure Traffic Manager (ZR) Networking • Azure Load Balancer (Z, ZR) • VPN gateway (ZR) • Azure ExpressRoute (ZR) • Azure Application Gateway (ZR) • Azure Firewall (ZR) • Azure Virtual WAN (ZR) Security • Azure Active Directory Domain Services (ZR) Storage • Azure Data Lake Storage (ZRS) • Azure Blob storage (ZRS, ZR) • Azure Managed Disks (Z) Additional capabilities • Azure Premium Files (ZRS) • Zone-redundant storage (ZRS) • Standard IP address (ZR) • Azure Traffic Analytics (ZR)

VMs supporting AZs: AV2-series, B-series, DSv2-series, DSv3-series, Dv2-series, Dv3-series, ESv3-series, Ev3-series, F-series, FS-series, FSv2-series, M-series.

For a list of Azure services that support Availability Zones, per Azure region, refer to the Availability Zones documentation.

SLA offered by Availability Zones

With Availability Zones, Azure offers industry best 99.99% VM uptime SLA. The full Azure SLA explains the guaranteed availability of Azure as a whole.

The following diagram illustrates the different levels of HA offered by a single VM, Availability Sets, and Availability Zones.

Using a VM workload as an example, a single VM has an SLA of 99.9%. This means the VM will be available 99.9% of the time. Within a single datacenter, the use of Availability Sets can increase the level of SLA to 99.95% by protecting a set of VMs, ensuring they will not all be on the same hardware. Within a region, VM workloads can be distributed across Availability Zones to increase the SLA to 99.99%. For more information, refer to Availability options for VMs in Azure.

Every organization has unique requirements, and you should design your applications to best meet your complex business needs. Defining a target SLA will make it possible to evaluate whether the architecture meets your business requirements. Some things to consider include:

- What are the availability requirements?
- How much downtime is acceptable?
- How much will potential downtime cost your business?
- How much should you invest in making the application highly available?
- What are the data backup requirements?
- What are the data replication requirements?
- What are the monitoring requirements?
- Does your application have specific latency requirements?

For additional guidance, refer to Principles of the reliability pillar.

Depending on the availability needs of an application, the cost and design complexity will vary. When building for a VM workload, there will be a cost associated with each VM. For example, two VMs per zone across three active zones will have a cost for six VMs. For pricing of VM workloads, refer to the Azure pricing calculator.

Next steps

- Azure Services that support Availability Zones
- Regions and Availability Zones in Azure
- Create a virtual machine in an availability zone using Azure CLI
- Create a virtual machine in an availability zone using Azure PowerShell
- Create a virtual machine in an availability zone using the Azure portal
- About Azure Edge Zone

Hybrid architecture design

11/19/2021 • 2 minutes to read • Edit Online

Many organizations need a hybrid approach to analytics, automation, and services because their data is hosted both on-premises and in the cloud. Organizations often extend on-premises data solutions to the cloud. To connect environments, organizations start by choosing a hybrid network architecture.

Learn about hybrid solutions

If you're new to Azure, the best place to start is with Microsoft Learn. Microsoft Learn is a free, online training platform that provides interactive learning for Microsoft products and more. The Introduction to Azure hybrid cloud services module helps you build foundational knowledge and takes you through core concepts.

Browse other hybrid solutions in Microsoft Learn

Path to production

Explore some options for connecting an on-premises network to Azure:

- Extend an on-premises network using VPN
- Extend an on-premises network using ExpressRoute
- Connect an on-premises network to Azure using ExpressRoute

Best practices

When you adopt a hybrid model, you can choose from multiple solutions to confidently deliver hybrid workloads. See these documents for information on running Azure data services anywhere, modernizing applications anywhere, and managing your workloads anywhere:

- Azure Automation in a hybrid environment
- Azure Arc hybrid management and deployment for Kubernetes clusters
- Run containers
- Use Azure file shares
- Back up files
- Manage workloads
- Monitor performance
- Disaster recovery for Azure Stack Hub VMs

Additional resources

The typical hybrid solution journey ranges from learning how to get started with a hybrid architecture to how to use Azure services in hybrid environments. However, you might also just be looking for additional reference and supporting material to help along the way for your specific situation. See these resources for general information on hybrid architectures:

- Browse hybrid and multicloud architectures
- Troubleshoot a hybrid VPN connection

Example solutions

Here are some example implementations to consider:

- Cross-cloud scaling
- Cross-platform chat
- Hybrid connections
- Unlock legacy data with Azure Stack

Overview of a hybrid workload

11/19/2021 • 4 minutes to read • Edit Online

Customer workloads are becoming increasingly complex, with many applications often running on different hardware across on-premises, multicloud, and the edge. Managing these disparate workload architectures, ensuring uncompromised security, and enabling developer agility are critical to success.

Azure uniquely helps you meet these challenges, giving you the flexibility to innovate anywhere in your hybrid environment while operating seamlessly and securely. The Well-Architected Framework includes a hybrid description for each of the five pillars: cost optimization, operational excellence, performance efficiency, reliability, and security. These descriptions create clarity on the considerations needed for your workloads to operate effectively across hybrid environments.

Adopting a hybrid model offers multiple solutions that enable you to confidently deliver hybrid workloads: run Azure data services anywhere, modernize applications anywhere, and manage your workloads anywhere.

Extend Azure management to any infrastructure

> **TIP**
>
> Applying the principles in this article series to each of your workloads will better prepare you for hybrid adoption. For larger or centrally managed organizations, hybrid and multicloud are commonly part of a broader strategic objective. If you need to scale these principle across a portfolio of workloads using hybrid and multicloud environments, you may want to start with the Cloud Adoption Framework's hybrid and multicloud scenario and best practices. Then return to this series to refine each of your workload architectures.

Use *Azure Arc enabled infrastructure* to extend Azure management to any infrastructure in a hybrid environment. Key features of Azure Arc enabled infrastructure are:

- **Unified Operations**
  - Organize resources such as virtual machines, Kubernetes clusters and Azure services deployed across your entire IT environment.
  - Manage and govern resources with a single pane of glass from Azure.
  - Integrate with Azure Lighthouse for managed service provider support.
- **Adopt cloud practices**
  - Easily adopt DevOps techniques such as infrastructure as code.
  - Empower developers with self-service and choice of tools.
  - Standardize change control with configuration management systems, such as GitOps and DSC.

Run Azure services anywhere

Azure Arc allows you to run Azure Services anywhere. This allows you to build consistent hybrid and multicloud application archtiectures by using Azure services that can run in Azure, on-premises, at the edge, or at other cloud providers.

Run Azure data services anywhere

Use *Azure Arc enabled data services* to run Azure data services anywhere to support your hybrid workloads. Key features of Azure Arc enabled data services are:

- Run Azure data services on any Kubernetes cluster deployed on any hardware.
- Gain cloud automation benefits, always up-to-date innovation in Azure data services, unified management of your on-premises and cloud data assets with a cloud billing model across both environments.
- Azure SQL Database and Azure PostgreSQL Hyperscale are the first set of Azure data services that are Azure Arc enabled.

Run Azure Application services anywhere

Use *Azure Arc enabled Application services* to run Azure App Service, Functions, Logic Apps, Event Grid, and API Management anywhere to support your hybrid workloads. Key features of Azure Arc enabled application services are as follows:

- Web Apps - Azure App Service makes building and managing web applications and APIs easy, with a fully managed platform and features like autoscaling, deployment slots, and integrated web authentication.
- Functions - Azure Functions makes event-driven programming simple, with state-of-the-art autoscaling, and with triggers and bindings to integrate with other Azure services.
- Logic Apps - Azure Logic Apps produces automated workflows for integrating apps, data, services, and backend systems, with a library of more than 400 connectors.
- Event Grid - Azure Event Grid simplifies event-based applications, with a single service for managing the routing of events from any source to any destination.
- Azure API Management gateway - Azure API Management provides a unified management experience and full observability across all internal and external APIs.

Modernize applications anywhere

Use the *Azure Stack family* to modernize applications without ever leaving the datacenter. Key features of the Azure Stack family are:

- Extend Azure to your on-premises workloads with Azure Stack Hub. Build and run cloud apps on premises, in connected or disconnected scenarios, to meet regulatory or technical requirements.
- Use Azure Stack HCI to run virtualized workloads on premises and easily connect to Azure to access cloud management and security services.
- Build and run your intelligent edge solutions on Azure Stack Edge, an Azure managed appliance to run machine learning models and compute at the edge to get results quickly—and close to where data is being generated. Easily transfer the full data set to Azure for further analysis or archive.

Manage workloads anywhere

Use *Azure Arc management* to extend Azure management to all assets in your workloads, regardless of where they are hosted. Key features of Azure Arc management are:

- **Adopt cloud practices**
  - Easily adopt DevOps techniques such as infrastructure as code.
  - Empower developers with self-service and choice of tools.
  - Standardize change control with configuration management systems, such as GitOps and DSC.
- **Scale across workloads with Unified Operations**
  - Organize resources such as virtual machines, Kubernetes clusters and Azure services deployed across your entire IT environment.
  - Manage and govern resources with a single pane of glass from Azure.
  - Integrate with Azure Lighthouse for managed service provider support.

Next steps

Configure hybrid cloud connectivity using Azure and Azure Stack Hub

11/19/2021 • 9 minutes to read • Edit Online

You can access resources with security in global Azure and Azure Stack Hub using the hybrid connectivity pattern.

In this solution, you'll build a sample environment to:

- Keep data on-premises to meet privacy or regulatory requirements but keep access to global Azure resources.
- Maintain a legacy system while using cloud-scaled app deployment and resources in global Azure.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that allows you to build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Prerequisites

A few components are required to build a hybrid connectivity deployment. Some of these components take time to prepare, so plan accordingly.

Azure

- If you don't have an Azure subscription, create a free account before you begin.
- Create a web app in Azure. Make note of the web app URL because you'll need it in the solution.

Azure Stack Hub

An Azure OEM/hardware partner can deploy a production Azure Stack Hub, and all users can deploy an Azure Stack Development Kit (ASDK).

- Use your production Azure Stack Hub or deploy the ASDK.

  NOTE

  Deploying the ASDK can take up to 7 hours, so plan accordingly.

- Deploy App Service PaaS services to Azure Stack Hub.
- Create plans and offers in the Azure Stack Hub environment.
- Create tenant subscription within the Azure Stack Hub environment.

Azure Stack Hub components

An Azure Stack Hub operator must deploy the App Service, create plans and offers, create a tenant subscription, and add the Windows Server 2016 image. If you already have these components, make sure they meet the requirements before you start this solution.

This solution example assumes that you have some basic knowledge of Azure and Azure Stack Hub. To learn more before starting the solution, read the following articles:

- Introduction to Azure
- Azure Stack Hub Key Concepts

Before you begin

Verify that you meet the following criteria before you start configuring hybrid cloud connectivity:

- You need an externally facing public IPv4 address for your VPN device. This IP address can't be located behind a NAT (Network Address Translation).
- All resources are deployed in the same region/location.

Solution example values

The examples in this solution use the following values. You can use these values to create a test environment or refer to them for a better understanding of the examples. For more information about VPN gateway settings, see About VPN Gateway Settings.

Connection specifications:

- **VPN type**: route-based
- **Connection type**: site-to-site (IPsec)
- **Gateway type**: VPN
- **Azure connection name**: Azure-Gateway-AzureStack-S2SGateway (the portal will autofill this value)
- **Azure Stack Hub connection name**: AzureStack-Gateway-Azure-S2SGateway (the portal will autofill this value)
- **Shared key**: any compatible with VPN hardware, with matching values on both sides of connection
- **Subscription**: any preferred subscription
- **Resource group**: Test-Infra

Network and subnet IP addresses:

AZURE/AZURE STACK HUB CONNECTION	NAME	SUBNET	IP ADDRESS
Azure vNet	ApplicationvNet 10.100.102.9/23	ApplicationSubnet 10.100.102.0/24	
		GatewaySubnet 10.100.103.0/24	
Azure Stack Hub vNet	ApplicationvNet 10.100.100.0/23	ApplicationSubnet 10.100.100.0/24	
		GatewaySubnet 10.100101.0/24	
Azure Virtual Network Gateway	Azure-Gateway		

AZURE/AZURE STACK HUB CONNECTION	NAME	SUBNET	IP ADDRESS
Azure Stack Hub Virtual Network Gateway	AzureStack-Gateway		
Azure Public IP	Azure-GatewayPublicIP		Determined at creation
Azure Stack Hub Public IP	AzureStack-GatewayPublicIP		Determined at creation
Azure Local Network Gateway	AzureStack-S2SGateway 10.100.100.0/23		Azure Stack Hub Public IP Value
Azure Stack Hub Local Network Gateway	Azure-S2SGateway 10.100.102.0/23		Azure Public IP Value

Create a virtual network in global Azure and Azure Stack Hub

Use the following steps to create a virtual network by using the portal. You can use these example values if you're using this article as only a solution. If you're using this article to configure a production environment, replace the example settings with your own values.

> **IMPORTANT**
>
> You must ensure that there isn't an overlap of IP addresses in Azure or Azure Stack Hub vNet address spaces.

To create a vNet in Azure:

1. Use your browser to connect to the Azure portal and sign in with your Azure account.
2. Select **Create a resource**. In the **Search the marketplace** field, enter 'virtual network'. Select **Virtual network** from the results.
3. From the **Select a deployment model** list, select **Resource Manager**, and then select **Create**.
4. On **Create virtual network**, configure the VNet settings. The required fields names are prefixed with a red asterisk. When you enter a valid value, the asterisk changes to a green check mark.

To create a vNet in Azure Stack Hub:

1. Repeat the steps above (1-4) using the Azure Stack Hub **tenant portal**.

Add a gateway subnet

Before connecting your virtual network to a gateway, you need to create the gateway subnet for the virtual network that you want to connect to. The gateway services use the IP addresses you specify in the gateway subnet.

In the Azure portal, navigate to the Resource Manager virtual network where you want to create a virtual network gateway.

1. Select the vNet to open the **Virtual network** page.

2. In **SETTINGS**, select **Subnets**.

3. On the **Subnets** page, select **+Gateway subnet** to open the **Add subnet** page.

4. The **Name** for the subnet is automatically filled in with the value 'GatewaySubnet'. This value is required for Azure to recognize the subnet as the gateway subnet.

5. Change the **Address range** values that are provided to match your configuration requirements and then select **OK**.

Create a Virtual Network Gateway in Azure and Azure Stack

Use the following steps to create a virtual network gateway in Azure.

1. On the left side of the portal page, select **+** and enter 'virtual network gateway' in the search field.

2. In **Results**, select **Virtual network gateway**.

3. In **Virtual network gateway**, select **Create** to open the **Create virtual network gateway** page.

4. On **Create virtual network gateway**, specify the values for your network gateway using our **Tutorial example values**. Include the following additional values:

   - **SKU**: basic
   - **Virtual Network**: Select the virtual network you created earlier. The gateway subnet you created is automatically selected.
   - **First IP Configuration**: The public IP of your gateway.
     - Select **Create gateway IP configuration**, which takes you to the **Choose public IP address** page.
     - Select **+Create new** to open the **Create public IP address** page.
     - Enter a **Name** for your public IP address. Leave the SKU as **Basic**, and then select **OK** to save your changes.

     > **NOTE**
     > Currently, VPN Gateway only supports Dynamic Public IP address allocation. However, this doesn't mean that the IP address changes after it's assigned to your VPN gateway. The only time the public IP address changes is when the gateway is deleted and re-created. Resizing, resetting, or other internal maintenance/upgrades to your VPN gateway don't change the IP address.

5. Verify your gateway settings.

6. Select **Create** to create the VPN gateway. The gateway settings are validated and the "Deploying Virtual network gateway" tile is shown on your dashboard.

   > **NOTE**
   > Creating a gateway can take up to 45 minutes. You may need to refresh your portal page to see the completed status.

   After the gateway is created, you can see the IP address assigned to it by looking at the virtual network in the portal. The gateway appears as a connected device. To see more information about the gateway, select the device.

7. Repeat the previous steps (1-5) on your Azure Stack Hub deployment.

Create the local network gateway in Azure and Azure Stack Hub

The local network gateway typically refers to your on-premises location. You give the site a name that Azure or Azure Stack Hub can refer to, and then specify:

- The IP address of the on-premises VPN device that you're creating a connection for.
- The IP address prefixes that will be routed through the VPN gateway to the VPN device. The address prefixes you specify are the prefixes located on your on-premises network.

> **NOTE**
> If your on-premises network changes or you need to change the public IP address for the VPN device, you can update these values later.

1. In the portal, select **+Create a resource**.
2. In the search box, enter **Local network gateway**, then select **Enter** to search. A list of results will display.
3. Select **Local network gateway**, then select **Create** to open the **Create local network gateway** page.
4. On **Create local network gateway**, specify the values for your local network gateway using our **Tutorial example values**. Include the following additional values:
   - **IP address**: The public IP address of the VPN device that you want Azure or Azure Stack Hub to connect to. Specify a valid public IP address that isn't behind a NAT so Azure can reach the address. If you don't have the IP address right now, you can use a value from the example as a placeholder. You'll have to go back and replace the placeholder with the public IP address of your VPN device. Azure can't connect to the device until you provide a valid address.
   - **Address Space**: the address range for the network that this local network represents. You can add multiple address space ranges. Make sure that the ranges you specify don't overlap with ranges of other networks that you want to connect to. Azure will route the address range that you specify to the on-premises VPN device IP address. Use your own values if you want to connect to your on-premises site, not an example value.
   - **Configure BGP settings**: Use only when configuring BGP. Otherwise, don't select this option.
   - **Subscription**: Verify that the correct subscription is showing.
   - **Resource Group**: Select the resource group that you want to use. You can either create a new resource group or select one that you've already created.
   - **Location**: Select the location that this object will be created in. You may want to select the same location that your VNet resides in, but you're not required to do so.
5. When you finish specifying the required values, select **Create** to create the local network gateway.
6. Repeat these steps (1-5) on your Azure Stack Hub deployment.

Configure your connection

Site-to-site connections to an on-premises network require a VPN device. The VPN device you configure is referred to as a connection. To configure your connection, you need:

- A shared key. This key is the same shared key that you specify when creating your site-to-site VPN connection. In our examples, we use a basic shared key. We recommend that you generate a more complex key to use.

- The public IP address of your virtual network gateway. You can view the public IP address by using the Azure portal, PowerShell, or CLI. To find the public IP address of your VPN gateway using the Azure portal, go to virtual network gateways, then select the name of your gateway.

Use the following steps to create a site-to-site VPN connection between your virtual network gateway and your on-premises VPN device.

1. In the Azure portal, select **+Create a resource**.
2. Search for **connections**.
3. In **Results**, select **Connections**.
4. On **Connection**, select **Create**.
5. On **Create Connection**, configure the following settings:
   - **Connection type**: Select site-to-site (IPSec).
   - **Resource Group**: Select your test resource group.
   - **Virtual Network Gateway**: Select the virtual network gateway you created.
   - **Local Network Gateway**: Select the local network gateway you created.
   - **Connection Name**: This name is autopopulated using the values from the two gateways.
   - **Shared Key**: This value must match the value that you're using for your local on-premises VPN device. The tutorial example uses 'abc123', but you should use something more complex. The important thing is that this value *must* be the same value that you specify when configuring your VPN device.
   - The values for **Subscription**, **Resource Group**, and **Location** are fixed.
6. Select **OK** to create your connection.

You can see the connection in the **Connections** page of the virtual network gateway. The status will go from *Unknown* to *Connecting*, and then to *Succeeded*.

Next steps

- To learn more about Azure Cloud Patterns, see Cloud Design Patterns.

Configure hybrid cloud identity for Azure and Azure Stack Hub apps

11/19/2021 • 2 minutes to read • Edit Online

Learn how to configure a hybrid cloud identity for your Azure and Azure Stack Hub apps.

You have two options for granting access to your apps in both global Azure and Azure Stack Hub.

- When Azure Stack Hub has a continuous connection to the internet, you can use Azure Active Directory (Azure AD).
- When Azure Stack Hub is disconnected from the internet, you can use Azure Directory Federated Services (AD FS).

You use service principals to grant access to your Azure Stack Hub apps for deployment or configuration using the Azure Resource Manager in Azure Stack Hub.

In this solution, you'll build a sample environment to:

- Establish a hybrid identity in global Azure and Azure Stack Hub
- Retrieve a token to access the Azure Stack Hub API.

You must have Azure Stack Hub operator permissions for the steps in this solution.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that lets you build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Create a service principal for Azure AD in the portal

If you deployed Azure Stack Hub using Azure AD as the identity store, you can create service principals just like you do for Azure. Use an app identity to access resources shows you how to perform the steps through the portal. Be sure you have the required Azure AD permissions before beginning.

Create a service principal for AD FS using PowerShell

If you deployed Azure Stack Hub with AD FS, you can use PowerShell to create a service principal, assign a role for access, and sign in from PowerShell using that identity. Use an app identity to access resources shows you how to perform the required steps using PowerShell.

Using the Azure Stack Hub API

The Azure Stack Hub API solution walks you through the process of retrieving a token to access the Azure Stack

Hub API.

Connect to Azure Stack Hub using PowerShell

The quickstart to get up and running with PowerShell in Azure Stack Hub walks you through the steps needed to install Azure PowerShell and connect to your Azure Stack Hub installation.

Prerequisites

You need an Azure Stack Hub installation connected to Azure AD with a subscription you can access. If you don't have an Azure Stack Hub installation, you can use these instructions to set up an Azure Stack Development Kit (ASDK).

Connect to Azure Stack Hub using code

To connect to Azure Stack Hub using code, use the Azure Resource Manager endpoints API to get the authentication and graph endpoints for your Azure Stack Hub installation. Then authenticate using REST requests. You can find a sample client application on GitHub.

> **NOTE**
>
> Unless the Azure SDK for your language of choice supports Azure API Profiles, the SDK may not work with Azure Stack Hub. To learn more about Azure API Profiles, see the manage API version profiles article.

Next steps

- To learn more about how identity is handled in Azure Stack Hub, see Identity architecture for Azure Stack Hub.
- To learn more about Azure Cloud Patterns, see Cloud Design Patterns.

Deploy an AI-based footfall detection solution using Azure and Azure Stack Hub

11/19/2021 • 7 minutes to read • Edit Online

This article describes how to deploy an AI-based solution that generates insights from real world actions by using Azure, Azure Stack Hub, and the Custom Vision AI Dev Kit.

In this solution, you learn how to:

- Deploy Cloud Native Application Bundles (CNAB) at the edge.
- Deploy an app that spans cloud boundaries.
- Use the Custom Vision AI Dev Kit for inference at the edge.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that allows you to build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Prerequisites

Before getting started with this deployment guide, make sure you:

- Review the Footfall detection pattern topic.
- Obtain user access to an Azure Stack Development Kit (ASDK) or Azure Stack Hub integrated system instance, with:
  - The Azure App Service on Azure Stack Hub resource provider installed. You need operator access to your Azure Stack Hub instance, or work with your administrator to install.
  - A subscription to an offer that provides App Service and Storage quota. You need operator access to create an offer.
- Obtain access to an Azure subscription.
  - If you don't have an Azure subscription, sign up for a free trial account before you begin.
- Create two service principals in your directory:
  - One set up for use with Azure resources, with access at the Azure subscription scope.
  - One set up for use with Azure Stack Hub resources, with access at the Azure Stack Hub subscription scope.
  - To learn more about creating service principals and authorizing access, see Use an app identity to access resources. If you prefer to use Azure CLI, see Create an Azure service principal with Azure CLI.
- Deploy Azure Cognitive Services in Azure or Azure Stack Hub.
  - First, learn more about Cognitive Services.
  - Then visit Deploy Azure Cognitive Services to Azure Stack Hub to deploy Cognitive Services on Azure

Stack Hub. You first need to sign up for access to the preview.

- Clone or download an unconfigured Azure Custom Vision AI Dev Kit. For details, see the Vision AI DevKit.
- Sign up for a Power BI account.
- An Azure Cognitive Services Face API subscription key and endpoint URL. You can get both with the Try Cognitive Services free trial. Or, follow the instructions in Create a Cognitive Services account.
- Install the following development resources:
  - Azure CLI 2.0
  - Docker CE
  - Porter. You use Porter to deploy cloud apps using CNAB bundle manifests that are provided for you.
  - Visual Studio Code
  - Azure IoT Tools for Visual Studio Code
  - Python extension for Visual Studio Code
  - Python

Deploy the hybrid cloud app

First you use the Porter CLI to generate a credential set, then deploy the cloud app.

1. Clone or download the repository containing the solution sample code.

2. Porter will generate a set of credentials that will automate deployment of the app. Before running the credential generation command, be sure to have the following available:

   - A service principal for accessing Azure resources, including the service principal ID, key, and tenant DNS.
   - The subscription ID for your Azure subscription.
   - A service principal for accessing Azure Stack Hub resources, including the service principal ID, key, and tenant DNS.
   - The subscription ID for your Azure Stack Hub subscription.
   - Your Azure Cognitive Services Face API key and resource endpoint URL.
3. Run the Porter credential generation process and follow the prompts:

   ```
   porter creds generate --tag intelligentedge/footfall-cloud-deployment:0.1.0
   ```

4. Porter also requires a set of parameters to run. Create a parameter text file and enter the following name/value pairs. Ask your Azure Stack Hub administrator if you need assistance with any of the required values.

   > **NOTE**
   > The `resource suffix` value is used to ensure that your deployment's resources have unique names across Azure. It must be a unique string of letters and numbers, no longer than 8 characters.

   ```
   azure_stack_tenant_arm="Your Azure Stack Hub tenant endpoint"
   azure_stack_storage_suffix="Your Azure Stack Hub storage suffix"
   azure_stack_keyvault_suffix="Your Azure Stack Hub keyVault suffix"
   resource_suffix="A unique string to identify your deployment"
   azure_location="A valid Azure region"
   azure_stack_location="Your Azure Stack Hub location identifier"
   powerbi_display_name="Your first and last name"
   powerbi_principal_name="Your Power BI account email address"
   ```

Save the text file and make a note of its path.

5. You're now ready to deploy the hybrid cloud app using Porter. Run the install command and watch as resources are deployed to Azure and Azure Stack Hub:

```
porter install footfall-cloud –tag intelligentedge/footfall-cloud-deployment:0.1.0 –creds footfall-cloud-deployment –param-file "path-to-cloud-parameters-file.txt"
```

6. Once deployment is complete, make note of the following values:

   - The camera's connection string.
   - The image storage account connection string.
   - The resource group names.

Prepare the Custom Vision AI DevKit

Next, set up the Custom Vision AI Dev Kit as shown in the Vision AI DevKit quickstart. You also set up and test your camera, using the connection string provided in the previous step.

Deploy the camera app

Use the Porter CLI to generate a credential set, then deploy the camera app.

1. Porter will generate a set of credentials that will automate deployment of the app. Before running the credential generation command, be sure to have the following available:

   - A service principal for accessing Azure resources, including the service principal ID, key, and tenant DNS.
   - The subscription ID for your Azure subscription.
   - The image storage account connection string provided when you deployed the cloud app.

2. Run the Porter credential generation process and follow the prompts:

```
porter creds generate --tag intelligentedge/footfall-camera-deployment:0.1.0
```

3. Porter also requires a set of parameters to run. Create a parameter text file and enter the following text. Ask your Azure Stack Hub administrator if you don't know some of the required values.

   > **NOTE**
   > The `deployment suffix` value is used to ensure that your deployment's resources have unique names across Azure. It must be a unique string of letters and numbers, no longer than 8 characters.

```
iot_hub_name="Name of the IoT Hub deployed"
deployment_suffix="Unique string here"
```

   Save the text file and make a note of its path.

4. You're now ready to deploy the camera app using Porter. Run the install command and watch as the IoT Edge deployment is created.

```
porter install footfall-camera –tag intelligentedge/footfall-camera-deployment:0.1.0 –creds footfall-camera-deployment –param-file "path-to-camera-parameters-file.txt"
```

5. Verify that the camera's deployment is complete by viewing the camera feed at `https://<camera-ip>:3000/`, where `<camara-ip>` is the camera IP address. This step may take up to 10 minutes.

Configure Azure Stream Analytics

Now that data is flowing to Azure Stream Analytics from the camera, we need to manually authorize it to communicate with Power BI.

1. From the Azure portal, open **All Resources**, and the *process-footfall[yoursuffix]* job.
2. In the **Job Topology** section of the Stream Analytics job pane, select the **Outputs** option.
3. Select the **traffic-output** output sink.
4. Select **Renew authorization** and sign in to your Power BI account.

5. Save the output settings.
6. Go to the **Overview** pane and select **Start** to start sending data to Power BI.
7. Select **Now** for job output start time and select **Start**. You can view the job status in the notification bar.

Create a Power BI Dashboard

1. Once the job succeeds, go to Power BI and sign in with your work or school account. If the Stream Analytics job query is outputting results, the *footfall-dataset* dataset you created exists under the **Datasets** tab.
2. From your Power BI workspace, select **+ Create** to create a new dashboard named *Footfall Analysis.*
3. At the top of the window, select **Add tile**. Then select **Custom Streaming Data** and **Next**. Choose the **footfall-dataset** under **Your Datasets**. Select **Card** from the **Visualization type** dropdown, and add **age** to **Fields**. Select **Next** to enter a name for the tile, and then select **Apply** to create the tile.
4. You can add additional fields and cards as desired.

Test Your Solution

Observe how the data in the cards you created in Power BI changes as different people walk in front of the camera. Inferences may take up to 20 seconds to appear once recorded.

Remove Your Solution

If you'd like to remove your solution, run the following commands using Porter, using the same parameter files that you created for deployment:

```
porter uninstall footfall-cloud -tag intelligentedge/footfall-cloud-deployment:0.1.0 -creds footfall-cloud-deployment -param-file "path-to-cloud-parameters-file.txt"

porter uninstall footfall-camera -tag intelligentedge/footfall-camera-deployment:0.1.0 -creds footfall-camera-deployment -param-file "path-to-camera-parameters-file.txt"
```

Next steps

- Learn more about Hybrid app design considerations
- Review and propose improvements to the code for this sample on GitHub.

Deploy an app that scales cross-cloud using Azure and Azure Stack Hub

11/19/2021 • 12 minutes to read • Edit Online

Learn how to create a cross-cloud solution to provide a manually triggered process for switching from an Azure Stack Hub hosted web app to an Azure hosted web app with autoscaling via traffic manager. This process ensures flexible and scalable cloud utility when under load.

With this pattern, your tenant may not be ready to run your app in the public cloud. However, it may not be economically feasible for the business to maintain the capacity required in their on-premises environment to handle spikes in demand for the app. Your tenant can make use of the elasticity of the public cloud with their on-premises solution.

In this solution, you'll build a sample environment to:

- Create a multi-node web app.
- Configure and manage the Continuous Deployment (CD) process.
- Publish the web app to Azure Stack Hub.
- Create a release.
- Learn to monitor and track your deployments.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that lets you build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Prerequisites

- Azure subscription. If needed, create a free account before beginning.
- An Azure Stack Hub integrated system or deployment of Azure Stack Development Kit (ASDK).
  - For instructions on installing Azure Stack Hub, see Install the ASDK.
  - For an ASDK post-deployment automation script, go to: https://github.com/mattmcspirit/azurestack
  - This installation may require a few hours to complete.
- Deploy App Service PaaS services to Azure Stack Hub.
- Create plans/offers within the Azure Stack Hub environment.
- Create tenant subscription within the Azure Stack Hub environment.
- Create a web app within the tenant subscription. Make note of the new web app URL for later use.
- Deploy Azure Pipelines virtual machine (VM) within the tenant subscription.
- Windows Server 2016 VM with .NET 3.5 is required. This VM will be built in the tenant subscription on Azure Stack Hub as the private build agent.

- Windows Server 2016 with SQL 2017 VM image is available in the Azure Stack Hub Marketplace. If this image isn't available, work with an Azure Stack Hub Operator to ensure it's added to the environment.

Issues and considerations

Scalability

The key component of cross-cloud scaling is the ability to deliver immediate and on-demand scaling between public and on-premises cloud infrastructure, providing consistent and reliable service.

Availability

Ensure locally deployed apps are configured for high-availability through on-premises hardware configuration and software deployment.

Manageability

The cross-cloud solution ensures seamless management and familiar interface between environments. PowerShell is recommended for cross-platform management.

Cross-cloud scaling

Get a custom domain and configure DNS

Update the DNS zone file for the domain. Azure AD will verify ownership of the custom domain name. Use Azure DNS for Azure/Microsoft 365/external DNS records within Azure, or add the DNS entry at a different DNS registrar.

1. Register a custom domain with a public registrar.
2. Sign in to the domain name registrar for the domain. An approved admin may be required to make DNS updates.
3. Update the DNS zone file for the domain by adding the DNS entry provided by Azure AD. (The DNS entry won't affect email routing or web hosting behaviors.)

Create a default multi-node web app in Azure Stack Hub

Set up hybrid continuous integration and continuous deployment (CI/CD) to deploy web apps to Azure and Azure Stack Hub and to autopush changes to both clouds.

> **NOTE**
>
> Azure Stack Hub with proper images syndicated to run (Windows Server and SQL) and App Service deployment are required. For more information, review the App Service documentation Prerequisites for deploying App Service on Azure Stack Hub.

Add Code to Azure Repos

Azure Repos

1. Sign in to Azure Repos with an account that has project creation rights on Azure Repos.

   Hybrid CI/CD can apply to both app code and infrastructure code. Use Azure Resource Manager templates for both private and hosted cloud development.

2. **Clone the repository** by creating and opening the default web app.

Create self-contained web app deployment for App Services in both clouds

1. Edit the **WebApplication.csproj** file. Select `Runtimeidentifier` and add `win10-x64`. (See Self-contained deployment documentation.)

2. Check in the code to Azure Repos using Team Explorer.

3. Confirm that the app code has been checked into Azure Repos.

Create the build definition

1. Sign in to Azure Pipelines to confirm the ability to create build definitions.

2. Add **-r win10-x64** code. This addition is necessary to trigger a self-contained deployment with .NET Core.

3. Run the build. The self-contained deployment build process will publish artifacts that run on Azure and Azure Stack Hub.

Use an Azure hosted agent

Using a hosted build agent in Azure Pipelines is a convenient option to build and deploy web apps. Maintenance and upgrades are done automatically by Microsoft Azure, enabling a continuous and uninterrupted development cycle.

Manage and configure the CD process

Azure Pipelines and Azure DevOps Services provide a highly configurable and manageable pipeline for releases to multiple environments like development, staging, QA, and production environments; including requiring approvals at specific stages.

Create release definition

1. Select the **plus** button to add a new release under the **Releases** tab in the **Build and Release** section of Azure DevOps Services.

2. Apply the Azure App Service Deployment template.

3. Under **Add artifact**, add the artifact for the Azure Cloud build app.

4. Under Pipeline tab, select the **Phase, Task** link of the environment and set the Azure cloud environment values.

5. Set the **environment name** and select the **Azure subscription** for the Azure Cloud endpoint.

6. Under **App service name**, set the required Azure app service name.

7. Enter "Hosted VS2017" under **Agent queue** for Azure cloud hosted environment.

8. In Deploy Azure App Service menu, select the valid **Package or Folder** for the environment. Select **OK** to **folder location**.

9. Save all changes and go back to **release pipeline**.

10. Add a new artifact selecting the build for the Azure Stack Hub app.

11. Add one more environment by applying the Azure App Service Deployment.

12. Name the new environment "Azure Stack".

13. Find the Azure Stack environment under **Task** tab.

14. Select the subscription for the Azure Stack endpoint.

15. Set the Azure Stack web app name as the App service name.

16. Select the Azure Stack agent.

17. Under the Deploy Azure App Service section, select the valid **Package or Folder** for the environment. Select **OK** to folder location.

18. Under Variable tab add a variable named `VSTS\_ARM\_REST\_IGNORE\_SSL\_ERRORS`, set its value as **true**, and scope to Azure Stack.

19. Select the **Continuous** deployment trigger icon in both artifacts and enable the **Continues** deployment trigger.

20. Select the **Pre-deployment** conditions icon in the Azure Stack environment and set the trigger to **After release.**

21. Save all changes.

> **NOTE**
>
> Some settings for the tasks may have been automatically defined as environment variables when creating a release definition from a template. These settings can't be modified in the task settings; instead, the parent environment item must be selected to edit these settings.

Publish to Azure Stack Hub via Visual Studio

By creating endpoints, an Azure DevOps Services build can deploy Azure Service apps to Azure Stack Hub. Azure Pipelines connects to the build agent, which connects to Azure Stack Hub.

1. Sign in to Azure DevOps Services and go to the app settings page.
2. On **Settings**, select **Security**.
3. In **VSTS Groups**, select **Endpoint Creators**.
4. On the **Members** tab, select **Add**.
5. In **Add users and groups**, enter a user name and select that user from the list of users.
6. Select **Save changes**.
7. In the **VSTS Groups** list, select **Endpoint Administrators**.
8. On the **Members** tab, select **Add**.
9. In **Add users and groups**, enter a user name and select that user from the list of users.
10. Select **Save changes**.

Now that the endpoint information exists, the Azure Pipelines to Azure Stack Hub connection is ready to use. The build agent in Azure Stack Hub gets instructions from Azure Pipelines and then the agent conveys endpoint information for communication with Azure Stack Hub.

Develop the app build

> **NOTE**
> Azure Stack Hub with proper images syndicated to run (Windows Server and SQL) and App Service deployment are required. For more information, see Prerequisites for deploying App Service on Azure Stack Hub.

Use Azure Resource Manager templates like web app code from Azure Repos to deploy to both clouds.

Add code to an Azure Repos project

1. Sign in to Azure Repos with an account that has project creation rights on Azure Stack Hub.
2. **Clone the repository** by creating and opening the default web app.

Create self-contained web app deployment for App Services in both clouds

1. Edit the **WebApplication.csproj** file: Select `Runtimeidentifier` and then add `win10-x64`. For more information, see Self-contained deployment documentation.
2. Use Team Explorer to check the code into Azure Repos.
3. Confirm that the app code was checked into Azure Repos.

Create the build definition

1. Sign in to Azure Pipelines with an account that can create a build definition.
2. Go to the **Build Web Application** page for the project.
3. In **Arguments**, add **-r win10-x64** code. This addition is required to trigger a self-contained deployment with .NET Core.
4. Run the build. The self-contained deployment build process will publish artifacts that can run on Azure and Azure Stack Hub.

Use an Azure hosted build agent

Using a hosted build agent in Azure Pipelines is a convenient option to build and deploy web apps. Maintenance and upgrades are done automatically by Microsoft Azure, enabling a continuous and uninterrupted development cycle.

Configure the continuous deployment (CD) process

Azure Pipelines and Azure DevOps Services provide a highly configurable and manageable pipeline for releases to multiple environments like development, staging, quality assurance (QA), and production. This process can include requiring approvals at specific stages of the app life cycle.

Create release definition

Creating a release definition is the final step in the app build process. This release definition is used to create a release and deploy a build.

1. Sign in to Azure Pipelines and go to **Build and Release** for the project.
2. On the **Releases** tab, select **[+]** and then pick **Create release definition**.
3. On **Select a Template**, choose **Azure App Service Deployment**, and then select **Apply**.
4. On **Add artifact**, from the **Source (Build definition)**, select the Azure Cloud build app.
5. On the **Pipeline** tab, select the **1 Phase, 1 Task** link to **View environment tasks**.
6. On the **Tasks** tab, enter Azure as the **Environment name** and select the AzureCloud Traders-Web EP from the **Azure subscription** list.
7. Enter the **Azure app service name**, which is `northwindtraders` in the next screen capture.

8. For the Agent phase, select **Hosted VS2017** from the **Agent queue** list.
9. In **Deploy Azure App Service**, select the valid **Package or folder** for the environment.
10. In **Select File or Folder**, select **OK** to **Location**.
11. Save all changes and go back to **Pipeline**.
12. On the **Pipeline** tab, select **Add artifact**, and choose the **NorthwindCloud Traders-Vessel** from the **Source (Build Definition)** list.
13. On **Select a Template**, add another environment. Pick **Azure App Service Deployment** and then select **Apply**.
14. Enter `Azure Stack Hub` as the **Environment name**.
15. On the **Tasks** tab, find and select Azure Stack Hub.
16. From the **Azure subscription** list, select **AzureStack Traders-Vessel EP** for the Azure Stack Hub endpoint.
17. Enter the Azure Stack Hub web app name as the **App service name**.
18. Under **Agent selection**, pick **AzureStack -b Douglas Fir** from the **Agent queue** list.
19. For **Deploy Azure App Service**, select the valid **Package or folder** for the environment. On **Select File Or Folder**, select **OK** for the folder **Location**.
20. On the **Variable** tab, find the variable named `VSTS\_ARM\_REST\_IGNORE\_SSL\_ERRORS`. Set the variable value to **true**, and set its scope to **Azure Stack Hub**.
21. On the **Pipeline** tab, select the **Continuous deployment trigger** icon for the NorthwindCloud Traders-Web artifact and set the **Continuous deployment trigger** to **Enabled**. Do the same thing for the **NorthwindCloud Traders-Vessel** artifact.
22. For the Azure Stack Hub environment, select the **Pre-deployment conditions** icon set the trigger to **After release**.
23. Save all changes.

> **NOTE**
>
> Some settings for release tasks are automatically defined as environment variables when creating a release definition from a template. These settings can't be modified in the task settings but can be modified in the parent environment items.

Create a release

1. On the **Pipeline** tab, open the **Release** list and select **Create release**.
2. Enter a description for the release, check to see that the correct artifacts are selected, and then select **Create**. After a few moments, a banner appears indicating that the new release was created and the release name is displayed as a link. Select the link to see the release summary page.
3. The release summary page shows details about the release. In the following screen capture for "Release-2", the **Environments** section shows the **Deployment status** for Azure as "IN PROGRESS", and the status for Azure Stack Hub is "SUCCEEDED". When the deployment status for the Azure environment changes to "SUCCEEDED", a banner appears indicating that the release is ready for approval. When a deployment is pending or has failed, a blue **(i)** information icon is shown. Hover over the icon to see a pop-up that contains the reason for delay or failure.

4. Other views, like the list of releases, also display an icon that indicates approval is pending. The pop-up for this icon shows the environment name and more details related to the deployment. It's easy for an admin see the overall progress of releases and see which releases are waiting for approval.

Monitor and track deployments

1. On the **Release-2** summary page, select **Logs**. During a deployment, this page shows the live log from the agent. The left pane shows the status of each operation in the deployment for each environment.

2. Select the person icon in the **Action** column for a pre-deployment or post-deployment approval to see who approved (or rejected) the deployment and the message they provided.

3. After the deployment finishes, the entire log file is displayed in the right pane. Select any **Step** in the left pane to see the log file for a single step, like **Initialize Job**. The ability to see individual logs makes it easier to trace and debug parts of the overall deployment. **Save** the log file for a step or **Download all logs as zip**.

4. Open the **Summary** tab to see general information about the release. This view shows details about the build, the environments it was deployed to, deployment status, and other information about the release.

5. Select an environment link (**Azure** or **Azure Stack Hub**) to see information about existing and pending deployments to a specific environment. Use these views as a quick way to check that the same build was deployed to both environments.

6. Open the **deployed production app** in a browser. For example, for the Azure App Services website, open the URL `https://[your-app-name\].azurewebsites.net`.

Integration of Azure and Azure Stack Hub provides a scalable cross-cloud solution

A flexible and robust multi-cloud service provides data security, back up and redundancy, consistent and rapid availability, scalable storage and distribution, and geo-compliant routing. This manually triggered process ensures reliable and efficient load switching between hosted web apps and immediate availability of crucial data.

Next steps

- To learn more about Azure Cloud Patterns, see Cloud Design Patterns.

Deploy a high availability Kubernetes cluster on Azure Stack Hub

11/19/2021 • 11 minutes to read • Edit Online

This article will show you how to build a highly available Kubernetes cluster environment, deployed on multiple Azure Stack Hub instances, in different physical locations.

In this solution deployment guide, you learn how to:

- Download and prepare the AKS Engine
- Connect to the AKS Engine Helper VM
- Deploy a Kubernetes cluster
- Connect to the Kubernetes cluster
- Connect Azure Pipelines to Kubernetes cluster
- Configure monitoring
- Deploy application
- Autoscale application
- Configure Traffic Manager
- Upgrade Kubernetes
- Scale Kubernetes

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that allows you to build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Prerequisites

Before getting started with this deployment guide, make sure you:

- Review the High availability Kubernetes cluster pattern article.
- Review the contents of the companion GitHub repository, which contains additional assets referenced in this article.
- Have an account that can access the Azure Stack Hub user portal, with at least "contributor" permissions.

Download and prepare AKS Engine

AKS Engine is a binary that can be used from any Windows or Linux host that can reach the Azure Stack Hub Azure Resource Manager endpoints. This guide describes deploying a new Linux (or Windows) VM on Azure Stack Hub. It will be used later when AKS Engine deploys the Kubernetes clusters.

> **NOTE**
> You can also use an existing Windows or Linux VM to deploy a Kubernetes cluster on Azure Stack Hub using AKS Engine.

The step-by-step process and requirements for AKS Engine are documented here:

- Install the AKS Engine on Linux in Azure Stack Hub (or using Windows)

AKS Engine is a helper tool to deploy and operate (unmanaged) Kubernetes clusters (in Azure and Azure Stack Hub).

The details and differences of AKS Engine on Azure Stack Hub are described here:

- What is the AKS Engine on Azure Stack Hub?
- AKS Engine on Azure Stack Hub (on GitHub)

The sample environment will use Terraform to automate the deployment of the AKS Engine VM. You can find the details and code in the companion GitHub repo.

The result of this step is a new resource group on Azure Stack Hub that contains the AKS Engine helper VM and related resources:

6 items Show hidden types

NAME	TYPE	LOCATION
aksengine-nic1	Network interface	seattle
aksengine-nsg	Network security group	seattle
aksengine-pip	Public IP address	seattle
aksengine-vm1	Virtual machine	seattle
aksengine-vm1-osdisk	Disk	seattle
aksengine-vnet	Virtual network	seattle

> **NOTE**
> If you have to deploy AKS Engine in a disconnected air-gapped environment, review Disconnected Azure Stack Hub Instances to learn more.

In the next step, we'll use the newly deployed AKS Engine VM to deploy a Kubernetes cluster.

Connect to the AKS Engine helper VM

First you must connect to the previously created AKS Engine helper VM.

The VM should have a Public IP Address and should be accessible via SSH (Port 22/TCP).

Connect Start Restart Stop Capture Delete Refresh

Resource group aksengine	Computer name (not available)
Status Running	Operating system Linux
Location seattle	Size Standard F2 (2 vcpus, 4 GiB memory)
Subscription AKSeArch	Public IP address 199.6.94.80

> **TIP**
>
> You can use a tool of your choice like MobaXterm, puTTY or PowerShell in Windows 10 to connect to a Linux VM using SSH.

```
ssh <username>@<ipaddress>
```

After connecting, run the command `aks-engine`. Go to Supported AKS Engine Versions to learn more about the AKS Engine and Kubernetes versions.

```
azureadmin@topicalredbird:~$ aks-engine
Usage:
  aks-engine [flags]
  aks-engine [command]

Available Commands:
  completion    Generates bash completion scripts
  deploy        Deploy an Azure Resource Manager template
  generate      Generate an Azure Resource Manager template
  get-versions  Display info about supported Kubernetes versions
  help          Help about any command
  rotate-certs  Rotate certificates on an existing Kubernetes cluster
  scale         Scale an existing Kubernetes cluster
  upgrade       Upgrade an existing Kubernetes cluster
  version       Print the version of AKS Engine

Flags:
      --debug               enable verbose debug logs
  -h, --help                help for aks-engine
      --show-default-model  Dump the default API model to stdout

Use "aks-engine [command] --help" for more information about a command.
```

Deploy a Kubernetes cluster

The AKS Engine helper VM itself hasn't created a Kubernetes cluster on our Azure Stack Hub, yet. Creating the cluster is the first action to take in the AKS Engine helper VM.

The step-by-step process is documented here:

- Deploy a Kubernetes cluster with the AKS engine on Azure Stack Hub

The end result of the `aks-engine deploy` command and the preparations in the previous steps is a fully featured Kubernetes cluster deployed into the tenant space of the first Azure Stack Hub instance. The cluster itself consists of Azure IaaS components like VMs, load balancers, VNets, disks, and so on.

1. Azure load balancer (K8s API Endpoint) 2) Worker Nodes (Agent Pool) 3) Master Nodes

The cluster is now up-and-running and in the next step we'll connect to it.

Connect to the Kubernetes cluster

You can now connect to the previously created Kubernetes cluster, either via SSH (using the SSH key specified as part of the deployment) or via `kubectl` (recommended). The Kubernetes command-line tool `kubectl` is available for Windows, Linux, and macOS here. It's already pre-installed and configured on the master nodes of our cluster:

```
ssh azureuser@<k8s-master-lb-ip>
```

```
azureuser@k8s-master-35064155-0:~$ kubectl
kubectl controls the Kubernetes cluster manager.

 Find more information at: https://kubernetes.io/docs/reference/kubectl/overview/

Basic Commands (Beginner):
  create         Create a resource from a file or from stdin.
  expose         Take a replication controller, service, deployment or pod and expose it as a new Kubernetes Service
  run            Run a particular image on the cluster
  set            Set specific features on objects
```

It's not recommended to use the master node as a jumpbox for administrative tasks. The `kubectl` configuration is stored in `.kube/config` on the master node(s) as well as on the AKS Engine VM. You can copy the configuration to an admin machine with connectivity to the Kubernetes cluster and use the `kubectl` command there. The `.kube/config` file is also used later to configure a service connection in Azure Pipelines.

> **IMPORTANT**
> Keep these files secure because they contain the credentials for your Kubernetes cluster. An attacker with access to the file has enough information to gain administrator access to it. All actions that are done using the initial `.kube/config` file are done using a cluster-admin account.

You can now try various commands using `kubectl` to check the status of your cluster. Here are example commands:

```
kubectl get nodes
```

```
NAME                      STATUS   ROLE     VERSION
k8s-linuxpool-35064155-0  Ready    agent    v1.14.8
k8s-linuxpool-35064155-1  Ready    agent    v1.14.8
k8s-linuxpool-35064155-2  Ready    agent    v1.14.8
k8s-master-35064155-0     Ready    master   v1.14.8
```

```
kubectl cluster-info
```

```
Kubernetes master is running at https://aks.***
CoreDNS is running at https://aks.***/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
kubernetes-dashboard is running at https://aks.***/api/v1/namespaces/kube-system/services/https:kubernetes-
dashboard:/proxy
Metrics-server is running at https://aks.***/api/v1/namespaces/kube-system/services/https:metrics-
server:/proxy
```

> **IMPORTANT**
> Kubernetes has its own **Role-based Access Control (RBAC)** model that allows you to create fine-grained role definitions and role bindings. This is the preferable way to control access to the cluster instead of handing out cluster-admin permissions.

Connect Azure Pipelines to Kubernetes clusters

To connect Azure Pipelines to the newly deployed Kubernetes cluster, we need its kubeconfig (`.kube/config`) file as explained in the previous step.

- Connect to one of the master nodes of your Kubernetes cluster.
- Copy the content of the `.kube/config` file.

- Go to Azure DevOps > Project Settings > Service Connections to create a new "Kubernetes" service connection (use KubeConfig as Authentication method)

> **IMPORTANT**
>
> Azure Pipelines (or its build agents) must have access to the Kubernetes API. If there is an Internet connection from Azure Pipelines to the Azure Stack Hub Kubernetes cluster, you'll need to deploy a self-hosted Azure Pipelines Build Agent.

When deploying self-hosted Agents for Azure Pipelines, you may deploy either on Azure Stack Hub, or on a machine with network connectivity to all required management endpoints. See the details here:

- Azure Pipelines agents on Windows or Linux

The pattern Deployment (CI/CD) considerations section contains a decision flow that helps you to understand whether to use Microsoft-hosted agents or self-hosted agents:

In this sample solution, the topology includes a self-hosted build agent on each Azure Stack Hub instance. The agent can access the Azure Stack Hub Management Endpoints and the Kubernetes cluster API endpoints.

This design fulfills a common regulatory requirement, which is to have only outbound connections from the application solution.

Configure monitoring

You can use Azure Monitor for containers to monitor the containers in the solution. This points Azure Monitor to the AKS Engine-deployed Kubernetes cluster on Azure Stack Hub.

There are two ways to enable Azure Monitor on your cluster. Both ways require you to set up a Log Analytics

workspace in Azure.

- Method one uses a Helm Chart
- Method two as part of the AKS Engine cluster specification

In the sample topology, "Method one" is used, which allows automation of the process and updates can be installed more easily.

For the next step, you need an Azure LogAnalytics Workspace (ID and Key), `Helm` (version 3), and `kubectl` on your machine.

Helm is a Kubernetes package manager, available as a binary that is runs on macOS, Windows, and Linux. It can be downloaded at helm.sh. Helm relies on the Kubernetes configuration file used for the `kubectl` command:

```
helm repo add incubator https://kubernetes-charts-incubator.storage.googleapis.com/
helm repo update

helm install incubator/azuremonitor-containers \
--set omsagent.secret.wsid=<your_workspace_id> \
--set omsagent.secret.key=<your_workspace_key> \
--set omsagent.env.clusterName=<my_prod_cluster> \
--generate-name
```

This command will install the Azure Monitor agent on your Kubernetes cluster:

```
kubectl get pods -n kube-system
```

```
NAME                                    READY   STATUS
omsagent-8qdm6                          1/1     Running
omsagent-r6ppm                          1/1     Running
omsagent-rs-76c45758f5-lmc4l            1/1     Running
```

The Operations Management Suite (OMS) Agent on your Kubernetes cluster will send monitoring data to your Azure Log Analytics Workspace (using outbound HTTPS). You can now use Azure Monitor to get deeper insights about your Kubernetes clusters on Azure Stack Hub. This design is a powerful way to demonstrate the power of analytics that can be automatically deployed with your application's clusters.

> **IMPORTANT**
>
> If Azure Monitor does not show any Azure Stack Hub data, please make sure that you have followed the instructions on how to add AzureMonitor-Containers solution to a Azure Log Analytics workspace carefully.

Deploy the application

Before installing our sample application, there's another step to configure the nginx-based Ingress controller on our Kubernetes cluster. The Ingress controller is used as a layer 7 load balancer to route traffic in our cluster based on host, path, or protocol. Nginx-ingress is available as a Helm Chart. For detailed instructions, refer to the Helm Chart GitHub repository.

Our sample application is also packaged as a Helm Chart, like the Azure Monitoring Agent in the previous step. As such, it's straightforward to deploy the application onto our Kubernetes cluster. You can find the Helm Chart files in the companion GitHub repo

The sample application is a three tier application, deployed onto a Kubernetes cluster on each of two Azure Stack Hub instances. The application uses a MongoDB database. You can learn more about how to get the data replicated across multiple instances in the pattern Data and Storage considerations.

After deploying the Helm Chart for the application, you'll see all three tiers of your application represented as deployments and stateful sets (for the database) with a single pod:

```
kubectl get pod,deployment,statefulset
```

```
NAME                                        READY   STATUS
pod/ratings-api-569d7f7b54-mrv5d            1/1     Running
pod/ratings-mongodb-0                       1/1     Running
pod/ratings-web-85667bfb86-l6vxz            1/1     Running

NAME                                        READY
deployment.extensions/ratings-api           1/1
deployment.extensions/ratings-web           1/1

NAME                                        READY
statefulset.apps/ratings-mongodb            1/1
```

On the services, side you'll find the nginx-based Ingress Controller and its public IP address:

```
kubectl get service
```

```
NAME                                        TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
kubernetes                                  ClusterIP      10.0.0.1       <none>        443/TCP
nginx-ingress-1588931383-controller         LoadBalancer   10.0.114.180   *public-ip*   443:30667/TCP
nginx-ingress-1588931383-default-backend    ClusterIP      10.0.76.54     <none>        80/TCP
ratings-api                                 ClusterIP      10.0.46.69     <none>        80/TCP
ratings-web                                 ClusterIP      10.0.161.124   <none>        80/TCP
```

The "External IP" address is our "application endpoint". It's how users will connect to open the application and will also be used as the endpoint for our next step Configure Traffic Manager.

Autoscale the application

You can optionally configure the Horizontal Pod Autoscaler to scale up or down based on certain metrics like CPU utilization. The following command will create a Horizontal Pod Autoscaler that maintains 1 to 10 replicas of the Pods controlled by the ratings-web deployment. HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 80%:

```
kubectl autoscale deployment ratings-web --cpu-percent=80 --min=1 --max=10
```

You may check the current status of autoscaler by running this command:

```
kubectl get hpa
```

```
NAME          REFERENCE                     TARGET    MINPODS   MAXPODS   REPLICAS   AGE
ratings-web   Deployment/ratings-web/scale  0% / 80%  1         10        1          18s
```

Configure Traffic Manager

To distribute traffic between two (or more) deployments of the application, we'll use Azure Traffic Manager. Azure Traffic Manager is a DNS-based traffic load balancer in Azure.

> **NOTE**
>
> Traffic Manager uses DNS to direct client requests to the most appropriate service endpoint, based on a traffic-routing method and the health of the endpoints.

Instead of using Azure Traffic Manager you can also use other global load-balancing solutions hosted on-premises. In the sample scenario, we'll use Azure Traffic Manager to distribute traffic between two instances of our application. They can run on Azure Stack Hub instances in the same or different locations:

In Azure, we configure Traffic Manager to point to the two different instances of our application:

As you can see, the two endpoints point to the two instances of the deployed application from the previous section.

At this point:

- The Kubernetes infrastructure has been created, including an ingress controller.
- Clusters have been deployed across two Azure Stack Hub instances.
- Monitoring has been configured.
- Azure Traffic Manager will load balance traffic across the two Azure Stack Hub instances.
- On top of this infrastructure, the sample three-tier application has been deployed in an automated way using Helm Charts.

The solution should now be up and accessible to users!

There are also some post-deployment operational considerations worth discussing, which are covered in the next two sections.

Upgrade Kubernetes

Consider the following topics when upgrading the Kubernetes cluster:

- Upgrading a Kubernetes cluster is a complex Day 2 operation that can be done using AKS Engine. For more information, see Upgrade a Kubernetes cluster on Azure Stack Hub.
- AKS Engine allows you to upgrade clusters to newer Kubernetes and base OS image versions. For more information, see Steps to upgrade to a newer Kubernetes version.
- You can also upgrade only the underlying nodes to newer base OS image versions. For more information, see Steps to only upgrade the OS image.

Newer base OS images contain security and kernel updates. It's the cluster operator's responsibility to monitor the availability of newer Kubernetes Versions and OS Images. The operator should plan and execute these upgrades using AKS Engine. The base OS images must be downloaded from the Azure Stack Hub Marketplace by the Azure Stack Hub Operator.

Scale Kubernetes

Scale is another Day 2 operation that can be orchestrated using AKS Engine.

The scale command reuses your cluster configuration file (apimodel.json) in the output directory, as input for a new Azure Resource Manager deployment. AKS Engine executes the scale operation against a specific agent pool. When the scale operation is complete, AKS Engine updates the cluster definition in that same apimodel.json file. The cluster definition reflects the new node count in order to reflect the updated, current cluster configuration.

- Scale a Kubernetes cluster on Azure Stack Hub

Next steps

- Learn more about Hybrid app design considerations.
- Review and propose improvements to the code for this sample on GitHub.

Deploy a highly available MongoDB solution across two Azure Stack Hub environments

11/19/2021 • 3 minutes to read • Edit Online

This article will step you through an automated deployment of a basic highly available (HA) MongoDB cluster with a disaster recovery (DR) site across two Azure Stack Hub environments. To learn more about MongoDB and high availability, see Replica Set Members.

In this solution, you'll create a sample environment to:

- Orchestrate a deployment across two Azure Stack Hubs.
- Use Docker to minimize dependency issues with Azure API profiles.
- Deploy a basic highly available MongoDB cluster with a disaster recovery site.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that lets you build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Architecture for MongoDB with Azure Stack Hub

Prerequisites for MongoDB with Azure Stack Hub

- Two connected Azure Stack Hub integrated systems (Azure Stack Hub). This deployment doesn't work on the Azure Stack Development Kit (ASDK). To learn more about Azure Stack Hub, see What is Azure Stack Hub?
  - A tenant subscription on each Azure Stack Hub.
  - **Make a note of each subscription ID and the Azure Resource Manager endpoint for each Azure Stack Hub.**
- An Azure Active Directory (Azure AD) service principal that has permissions to the tenant subscription on each Azure Stack Hub. You may need to create two service principals if the Azure Stack Hubs are deployed against different Azure AD tenants. To learn how to create a service principal for Azure Stack Hub, see Use an

app identity to access Azure Stack Hub resources.

  - **Make a note of each service principal's application ID, client secret, and tenant name (xxxxx.onmicrosoft.com).**
- Ubuntu 16.04 syndicated to each Azure Stack Hub's Marketplace. To learn more about marketplace syndication, see Download Marketplace items to Azure Stack Hub.
- Docker for Windows installed on your local machine.

Get the Docker image

Docker images for each deployment eliminate dependency issues between different versions of Azure PowerShell.

1. Make sure that Docker for Windows is using Windows containers.

2. Run the following command in an elevated command prompt to get the Docker container with the deployment scripts:

```
docker pull intelligentedge/mongodb-hadr:1.0.0
```

Deploy the clusters

1. Once the container image has been successfully pulled, start the image:

```
docker run -it intelligentedge/mongodb-hadr:1.0.0 powershell
```

2. Once the container has started, you'll be given an elevated PowerShell terminal in the container. Change directories to get to the deployment script:

```
cd .\MongoHADRDemo\
```

3. Run the deployment. Provide credentials and resource names where needed. HA refers to the Azure Stack Hub where the HA cluster will be deployed. DR refers to the Azure Stack Hub where the DR cluster will be deployed:

```
.\Deploy-AzureResourceGroup.ps1 `
-AzureStackApplicationId_HA "applicationIDforHAServicePrincipal" `
-AzureStackApplicationSercet_HA "clientSecretforHAServicePrincipal" `
-AADTenantName_HA "hatenantname.onmicrosoft.com" `
-AzureStackResourceGroup_HA "haresourcegroupname" `
-AzureStackArmEndpoint_HA "https://management.haazurestack.com" `
-AzureStackSubscriptionId_HA "haSubscriptionId" `
-AzureStackApplicationId_DR "applicationIDforDRServicePrincipal" `
-AzureStackApplicationSercet_DR "ClientSecretforDRServicePrincipal" `
-AADTenantName_DR "drtenantname.onmicrosoft.com" `
-AzureStackResourceGroup_DR "drresourcegroupname" `
-AzureStackArmEndpoint_DR "https://management.drazurestack.com" `
-AzureStackSubscriptionId_DR "drSubscriptionId"
```

4. Type `Y` to allow the NuGet provider to be installed, which will kick off the API Profile "2018-03-01-hybrid" modules to be installed.

5. The HA resources will deploy first. Monitor the deployment and wait for it to finish. Once you have the message stating that the HA deployment is finished, you can check the HA Azure Stack Hub's portal to see the resources deployed.

6. Continue with the deployment of DR resources and decide if you'd like to enable a jump box on the DR Azure Stack Hub to interact with the cluster.

7. Wait for DR resource deployment to finish.

8. Once DR resource deployment has finished, exit the container:

```
exit
```

Next steps

- If you enabled the jump box VM on the DR Azure Stack Hub, you can connect via SSH and interact with the MongoDB cluster by installing the mongo CLI. To learn more about interacting with MongoDB, see The mongo Shell.
- To learn more about hybrid cloud apps, see Hybrid Cloud Solutions..
- Modify the code to this sample on GitHub.

Deploy hybrid app with on-premises data that scales cross-cloud

11/19/2021 • 17 minutes to read • Edit Online

This solution guide shows you how to deploy a hybrid app that spans both Azure and Azure Stack Hub and uses a single on-premises data source.

By using a hybrid cloud solution, you can combine the compliance benefits of a private cloud with the scalability of the public cloud. Your developers can also take advantage of the Microsoft developer ecosystem and apply their skills to the cloud and on-premises environments.

Overview and assumptions

Follow this tutorial to set up a workflow that lets developers deploy an identical web app to a public cloud and a private cloud. This app can access a non-internet routable network hosted on the private cloud. These web apps are monitored and when there's a spike in traffic, a program modifies the DNS records to redirect traffic to the public cloud. When traffic drops to the level before the spike, traffic is routed back to the private cloud.

This tutorial covers the following tasks:

- Deploy a hybrid-connected SQL Server database server.
- Connect a web app in global Azure to a hybrid network.
- Configure DNS for cross-cloud scaling.
- Configure SSL certificates for cross-cloud scaling.
- Configure and deploy the web app.
- Create a Traffic Manager profile and configure it for cross-cloud scaling.
- Set up Application Insights monitoring and alerting for increased traffic.
- Configure automatic traffic switching between global Azure and Azure Stack Hub.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that allows you to build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Assumptions

This tutorial assumes that you have a basic knowledge of global Azure and Azure Stack Hub. If you want to learn more before starting the tutorial, review these articles:

- Introduction to Azure
- Azure Stack Hub Key Concepts

This tutorial also assumes that you have an Azure subscription. If you don't have a subscription, create a free account before you begin.

Prerequisites

Before you start this solution, make sure you meet the following requirements:

- An Azure Stack Development Kit (ASDK) or a subscription on an Azure Stack Hub Integrated System. To deploy the ASDK, follow the instructions in Deploy the ASDK using the installer.
- Your Azure Stack Hub installation should have the following installed:
  - The Azure App Service. Work with your Azure Stack Hub Operator to deploy and configure the Azure App Service on your environment. This tutorial requires the App Service to have at least one (1) available dedicated worker role.
  - A Windows Server 2016 image.
  - A Windows Server 2016 with a Microsoft SQL Server image.
  - The appropriate plans and offers.
  - A domain name for your web app. If you don't have a domain name, you can buy one from a domain provider like GoDaddy, Bluehost, and InMotion.
- An SSL certificate for your domain from a trusted certificate authority like LetsEncrypt.
- A web app that communicates with a SQL Server database and supports Application Insights. You can download the dotnetcore-sqldb-tutorial sample app from GitHub.
- A hybrid network between an Azure virtual network and Azure Stack Hub virtual network. For detailed instructions, see Configure hybrid cloud connectivity with Azure and Azure Stack Hub.
- A hybrid continuous integration/continuous deployment (CI/CD) pipeline with a private build agent on Azure Stack Hub. For detailed instructions, see Configure hybrid cloud identity with Azure and Azure Stack Hub apps.

Deploy a hybrid-connected SQL Server database server

1. Sign to the Azure Stack Hub user portal.
2. On the **Dashboard**, select **Marketplace**.

3. In **Marketplace**, select **Compute**, and then choose **More**. Under **More**, select the **Free SQL Server License: SQL Server 2017 Developer on Windows Server** image.

4. On **Free SQL Server License: SQL Server 2017 Developer on Windows Server**, select **Create**.

5. On **Basics > Configure basic settings**, provide a **Name** for the virtual machine (VM), a **User name** for the SQL Server SA, and a **Password** for the SA. From the **Subscription** drop-down list, select the subscription that you're deploying to. For **Resource group**, use **Choose existing** and put the VM in the same resource group as your Azure Stack Hub web app.

6. Under **Size**, pick a size for your VM. For this tutorial, we recommend A2_Standard or a DS2_V2_Standard.
7. Under **Settings > Configure optional features**, configure the following settings:
   - **Storage account**: Create a new account if you need one.
   - **Virtual network**:

     > **IMPORTANT**
     > Make sure your SQL Server VM is deployed on the same virtual network as the VPN gateways.

   - **Public IP address**: Use the default settings.
   - **Network security group**: (NSG). Create a new NSG.
   - **Extensions and Monitoring**: Keep the default settings.
   - **Diagnostics storage account**: Create a new account if you need one.
   - Select **OK** to save your configuration.

8. Under **SQL Server settings**, configure the following settings:

   - For **SQL connectivity**, select **Public (Internet)**.
   - For **Port**, keep the default, **1433**.
   - For **SQL authentication**, select **Enable**.

   > **NOTE**
   > When you enable SQL authentication, it should auto-populate with the "SQLAdmin" information that you configured in **Basics**.

   - For the rest of the settings, keep the defaults. Select **OK**.

9. On **Summary**, review the VM configuration and then select **OK** to start the deployment.

10. It takes some time to create the new VM. You can view the STATUS of your VMs in **Virtual machines**.

Create web apps in Azure and Azure Stack Hub

The Azure App Service simplifies running and managing a web app. Because Azure Stack Hub is consistent with Azure, the App Service can run in both environments. You'll use the App Service to host your app.

Create web apps

1. Create a web app in Azure by following the instructions in Manage an App Service plan in Azure. Make sure you put the web app in the same subscription and resource group as your hybrid network.

2. Repeat the previous step (1) in Azure Stack Hub.

Add route for Azure Stack Hub

The App Service on Azure Stack Hub must be routable from the public internet to let users access your app. If your Azure Stack Hub is accessible from the internet, make a note of the public-facing IP address or URL for the Azure Stack Hub web app.

If you're using an ASDK, you can configure a static NAT mapping to expose App Service outside the virtual environment.

Connect a web app in Azure to a hybrid network

To provide connectivity between the web front end in Azure and the SQL Server database in Azure Stack Hub, the web app must be connected to the hybrid network between Azure and Azure Stack Hub. To enable connectivity, you'll have to:

- Configure point-to-site connectivity.
- Configure the web app.
- Modify the local network gateway in Azure Stack Hub.

Configure the Azure virtual network for point-to-site connectivity

The virtual network gateway in the Azure side of the hybrid network must allow point-to-site connections to integrate with Azure App Service.

1. In the Azure portal, go to the virtual network gateway page. Under **Settings**, select **Point-to-site configuration**.

2. Select **Configure now** to configure point-to-site.

3. On the **Point-to-site** configuration page, enter the private IP address range that you want to use in **Address pool**.

> **NOTE**
>
> Make sure that the range you specify doesn't overlap with any of the address ranges already used by subnets in the global Azure or Azure Stack Hub components of the hybrid network.

Under **Tunnel Type**, uncheck the **IKEv2 VPN**. Select **Save** to finish configuring point-to-site.

Integrate the Azure App Service app with the hybrid network

1. To connect the app to the Azure VNet, follow the instructions in Gateway required VNet integration.

2. Go to **Settings** for the App Service plan hosting the web app. In **Settings**, select **Networking**.

3. In **VNET Integration**, select **Click here to manage**.

4. Select the VNET that you want to configure. Under **IP ADDRESSES ROUTED TO VNET**, enter the IP address range for the Azure VNet, the Azure Stack Hub VNet, and the point-to-site address spaces. Select **Save** to validate and save these settings.

To learn more about how App Service integrates with Azure VNets, see Integrate your app with an Azure Virtual Network.

Configure the Azure Stack Hub virtual network

The local network gateway in the Azure Stack Hub virtual network needs to be configured to route traffic from the App Service point-to-site address range.

1. In the Azure Stack Hub portal, go to **Local network gateway**. Under **Settings**, select **Configuration**.

2. In **Address space**, enter the point-to-site address range for the virtual network gateway in Azure.

3. Select **Save** to validate and save the configuration.

Configure DNS for cross-cloud scaling

By properly configuring DNS for cross-cloud apps, users can access the global Azure and Azure Stack Hub instances of your web app. The DNS configuration for this tutorial also lets Azure Traffic Manager route traffic when the load increases or decreases.

This tutorial uses Azure DNS to manage the DNS because App Service domains won't work.

Create subdomains

Because Traffic Manager relies on DNS CNAMEs, a subdomain is needed to properly route traffic to endpoints. For more information about DNS records and domain mapping, see map domains with Traffic Manager.

For the Azure endpoint, you'll create a subdomain that users can use to access your web app. For this tutorial, can use **app.northwind.com**, but you should customize this value based on your own domain.

You'll also need to create a subdomain with an A record for the Azure Stack Hub endpoint. You can use **azurestack.northwind.com**.

Configure a custom domain in Azure

1. Add the **app.northwind.com** hostname to the Azure web app by mapping a CNAME to Azure App Service.

Configure custom domains in Azure Stack Hub

1. Add the **azurestack.northwind.com** hostname to the Azure Stack Hub web app by mapping an A record to Azure App Service. Use the internet-routable IP address for the App Service app.

2. Add the **app.northwind.com** hostname to the Azure Stack Hub web app by mapping a CNAME to Azure App Service. Use the hostname you configured in the previous step (1) as the target for the CNAME.

Configure SSL certificates for cross-cloud scaling

It's important to ensure sensitive data collected by your web app is secure in transit to and when stored on the SQL database.

You'll configure your Azure and Azure Stack Hub web apps to use SSL certificates for all incoming traffic.

Add SSL to Azure and Azure Stack Hub

To add SSL to Azure:

1. Make sure that the SSL certificate you get is valid for the subdomain you created. (It's okay to use wildcard certificates.)
2. In the Azure portal, follow the instructions in the **Prepare your web app** and **Bind your SSL certificate** sections of the Bind an existing custom SSL certificate to Azure Web Apps article. Select **SNI-based SSL** as the **SSL Type**.
3. Redirect all traffic to the HTTPS port. Follow the instructions in the **Enforce HTTPS** section of the Bind an existing custom SSL certificate to Azure Web Apps article.

To add SSL to Azure Stack Hub:

1. Repeat steps 1-3 that you used for Azure, using the Azure Stack Hub portal.

Configure and deploy the web app

You'll configure the app code to report telemetry to the correct Application Insights instance and configure the web apps with the right connection strings. To learn more about Application Insights, see What is Application Insights?

Add Application Insights

1. Open your web app in Microsoft Visual Studio.
2. Add Application Insights to your project to transmit the telemetry that Application Insights uses to create alerts when web traffic increases or decreases.

Configure dynamic connection strings

Each instance of the web app will use a different method to connect to the SQL database. The app in Azure uses the private IP address of the SQL Server VM and the app in Azure Stack Hub uses the public IP address of the SQL Server VM.

> **NOTE**
>
> On an Azure Stack Hub integrated system, the public IP address shouldn't be internet-routable. On an ASDK, the public IP address isn't routable outside the ASDK.

You can use App Service environment variables to pass a different connection string to each instance of the app.

1. Open the app in Visual Studio.
2. Open Startup.cs and find the following code block:

```
services.AddDbContext<MyDatabaseContext>(options =>
    options.UseSqlite("Data Source=localdatabase.db"));
```

3. Replace the previous code block with the following code, which uses a connection string defined in the *appsettings.json* file:

```
services.AddDbContext<MyDatabaseContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("MyDbConnection")));
 // Automatically perform database migration
 services.BuildServiceProvider().GetService<MyDatabaseContext>().Database.Migrate();
```

Configure App Service app settings

1. Create connection strings for Azure and Azure Stack Hub. The strings should be the same, except for the IP addresses that are used.

2. In Azure and Azure Stack Hub, add the appropriate connection string as an app setting in the web app, using `SQLCONNSTR\_` as a prefix in the name.

3. **Save** the web app settings and restart the app.

Enable automatic scaling in global Azure

When you create your web app in an App Service environment, it starts with one instance. You can automatically scale out to add instances to provide more compute resources for your app. Similarly, you can automatically scale in and reduce the number of instances your app needs.

> **NOTE**
> You need to have an App Service plan to configure scale out and scale in. If you don't have a plan, create one before starting the next steps.

Enable automatic scale-out

1. In the Azure portal, find the App Service plan for the sites you want to scale out, and then select **Scale-out (App Service plan)**.

2. Select **Enable autoscale**.

3. Enter a name for **Autoscale Setting Name**. For the **Default** auto scale rule, select **Scale based on a metric**. Set the **Instance limits** to **Minimum: 1**, **Maximum: 10**, and **Default: 1**.

4. Select **+Add a rule**.

5. In **Metric Source**, select **Current Resource**. Use the following Criteria and Actions for the rule.

Criteria

1. Under **Time Aggregation,** select **Average**.

2. Under **Metric Name**, select **CPU Percentage**.

3. Under **Operator**, select **Greater than**.

   - Set the **Threshold** to **50**.
   - Set the **Duration** to **10**.

Action

1. Under **Operation**, select **Increase Count by**.

2. Set the **Instance Count** to **2**.

3. Set the **Cool down** to **5**.

4. Select **Add**.

5. Select the **+ Add a rule**.

6. In **Metric Source**, select **Current Resource.**

> **NOTE**
>
> The current resource will contain your App Service plan's name/GUID and the **Resource Type** and **Resource** drop-down lists will be unavailable.

Enable automatic scale in

When traffic decreases, the Azure web app can automatically reduce the number of active instances to reduce costs. This action is less aggressive than scale-out and minimizes the impact on app users.

1. Go to the **Default** scale out condition, then select **+ Add a rule**. Use the following Criteria and Actions for the rule.

Criteria

1. Under **Time Aggregation,** select **Average**.
2. Under **Metric Name**, select **CPU Percentage**.
3. Under **Operator**, select **Less than**.
   - Set the **Threshold** to **30**.
   - Set the **Duration** to **10**.

Action

1. Under **Operation**, select **Decrease Count by**.
   - Set the **Instance Count** to **1**.
   - Set the **Cool down** to **5**.
2. Select **Add**.

Create a Traffic Manager profile and configure cross-cloud scaling

Create a Traffic Manager profile using the Azure portal, then configure endpoints to enable cross-cloud scaling.

Create Traffic Manager profile

1. Select **Create a resource**.
2. Select **Networking**.
3. Select **Traffic Manager profile** and configure the following settings:
   - In **Name**, enter a name for your profile. This name **must** be unique in the trafficmanager.net zone and is used to create a new DNS name (for example, northwindstore.trafficmanager.net).
   - For **Routing method**, select the **Weighted**.
   - For **Subscription**, select the subscription you want to create this profile in.
   - In **Resource Group**, create a new resource group for this profile.
   - In **Resource group location**, select the location of the resource group. This setting refers to the location of the resource group and has no impact on the Traffic Manager profile that's deployed globally.
4. Select **Create**.

When the global deployment of your Traffic Manager profile is complete, it's shown in the list of

resources for the resource group you created it under.

Add Traffic Manager endpoints

1. Search for the Traffic Manager profile you created. If you navigated to the resource group for the profile, select the profile.
2. In **Traffic Manager profile**, under **SETTINGS**, select **Endpoints**.
3. Select **Add**.
4. In **Add endpoint**, use the following settings for Azure Stack Hub:
   - For **Type**, select **External endpoint**.
   - Enter a **Name** for the endpoint.
   - For **Fully qualified domain name (FQDN) or IP**, enter the external URL for your Azure Stack Hub web app.
   - For **Weight**, keep the default, **1**. This weight results in all traffic going to this endpoint if it's healthy.
   - Leave **Add as disabled** unchecked.
5. Select **OK** to save the Azure Stack Hub endpoint.

You'll configure the Azure endpoint next.

1. On **Traffic Manager profile**, select **Endpoints**.
2. Select **+Add**.
3. On **Add endpoint**, use the following settings for Azure:
   - For **Type**, select **Azure endpoint**.
   - Enter a **Name** for the endpoint.
   - For **Target resource type**, select **App Service**.
   - For **Target resource**, select **Choose an app service** to see a list of Web Apps in the same subscription.
   - In **Resource**, pick the App service that you want to add as the first endpoint.
   - For **Weight**, select **2**. This setting results in all traffic going to this endpoint if the primary endpoint is unhealthy, or if you have a rule/alert that redirects traffic when triggered.
   - Leave **Add as disabled** unchecked.
4. Select **OK** to save the Azure endpoint.

After both endpoints are configured, they're listed in **Traffic Manager profile** when you select **Endpoints**. The example in the following screen capture shows two endpoints, with status and configuration information for each one.

Set up Application Insights monitoring and alerting in Azure

Azure Application Insights lets you monitor your app and send alerts based on conditions you configure. Some examples are: the app is unavailable, is experiencing failures, or is showing performance issues.

You'll use Azure Application Insights metrics to create alerts. When these alerts trigger, your web app's instance will automatically switch from Azure Stack Hub to Azure to scale out, and then back to Azure Stack Hub to scale in.

Create an alert from metrics

In the Azure portal, go to the resource group for this tutorial, and select the Application Insights instance to open **Application Insights**.

You'll use this view to create a scale-out alert and a scale-in alert.

Create the scale-out alert

1. Under **CONFIGURE**, select **Alerts (classic)**.
2. Select **Add metric alert (classic)**.
3. In **Add rule**, configure the following settings:
   - For **Name**, enter **Burst into Azure Cloud**.
   - A **Description** is optional.
   - Under **Source** > **Alert on**, select **Metrics**.
   - Under **Criteria**, select your subscription, the resource group for your Traffic Manager profile, and the name of the Traffic Manager profile for the resource.
4. For **Metric**, select **Request Rate**.
5. For **Condition**, select **Greater than**.
6. For **Threshold**, enter **2**.
7. For **Period**, select **Over the last 5 minutes**.
8. Under **Notify via**:
   - Check the checkbox for **Email owners, contributors, and readers**.
   - Enter your email address for **Additional administrator email(s)**.
9. On the menu bar, select **Save**.

Create the scale-in alert

1. Under **CONFIGURE**, select **Alerts (classic)**.
2. Select **Add metric alert (classic)**.
3. In **Add rule**, configure the following settings:
   - For **Name**, enter **Scale back into Azure Stack Hub**.

- A **Description** is optional.
- Under **Source** > **Alert on**, select **Metrics**.
- Under **Criteria**, select your subscription, the resource group for your Traffic Manager profile, and the name of the Traffic Manager profile for the resource.

4. For **Metric**, select **Request Rate**.
5. For **Condition**, select **Less than**.
6. For **Threshold**, enter **2**.
7. For **Period**, select **Over the last 5 minutes**.
8. Under **Notify via**:
   - Check the checkbox for **Email owners, contributors, and readers**.
   - Enter your email address for **Additional administrator email(s)**.
9. On the menu bar, select **Save**.

The following screenshot shows the alerts for scale-out and scale-in.

Redirect traffic between Azure and Azure Stack Hub

You can configure manual or automatic switching of your web app traffic between Azure and Azure Stack Hub.

Configure manual switching between Azure and Azure Stack Hub

When your web site reaches the thresholds that you configure, you'll receive an alert. Use the following steps to manually redirect traffic to Azure.

1. In the Azure portal, select your Traffic Manager profile.

2. Select **Endpoints**.
3. Select the **Azure endpoint**.
4. Under **Status**, select **Enabled**, and then select **Save**.

5. On **Endpoints** for the Traffic Manager profile, select **External endpoint**.

6. Under **Status**, select **Disabled**, and then select **Save**.

After the endpoints are configured, app traffic goes to your Azure scale-out web app instead of the Azure Stack Hub web app.

Search endpoints

NAME	STATUS	MONITOR STATUS	TYPE	WEIGHT
NorthwindCloudShop-AzureStack	Disabled	Disabled	External endpoint	1
NorthwindCloudShop-Azure	Enabled	Online	Azure endpoint	2

To reverse the flow back to Azure Stack Hub, use the previous steps to:

- Enable the Azure Stack Hub endpoint.
- Disable the Azure endpoint.

Configure automatic switching between Azure and Azure Stack Hub

You can also use Application Insights monitoring if your app runs in a serverless environment provided by Azure Functions.

In this scenario, you can configure Application Insights to use a webhook that calls a function app. This app automatically enables or disables an endpoint in response to an alert.

Use the following steps as a guide to configure automatic traffic switching.

1. Create an Azure Function app.
2. Create an HTTP-triggered function.
3. Import the Azure SDKs for Resource Manager, Web Apps, and Traffic Manager.
4. Develop code to:
   - Authenticate to your Azure subscription.
   - Use a parameter that toggles the Traffic Manager endpoints to direct traffic to Azure or Azure Stack Hub.
5. Save your code and add the function app's URL with the appropriate parameters to the **Webhook** section of the Application Insights alert rule settings.
6. Traffic is automatically redirected when an Application Insights alert fires.

Next steps

- To learn more about Azure Cloud Patterns, see Cloud Design Patterns.

Deploy a SQL Server 2016 availability group across two Azure Stack Hub environments

11/19/2021 • 3 minutes to read • Edit Online

This article will step you through an automated deployment of a basic highly available (HA) SQL Server 2016 Enterprise cluster with an asynchronous disaster recovery (DR) site across two Azure Stack Hub environments. To learn more about SQL Server 2016 and high availability, see Always On availability groups: a high-availability and disaster-recovery solution.

In this solution, you'll build a sample environment to:

- Orchestrate a deployment across two Azure Stack Hubs.
- Use Docker to minimize dependency issues with Azure API profiles.
- Deploy a basic highly available SQL Server 2016 Enterprise cluster with a disaster recovery site.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that lets you build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Architecture for SQL Server 2016

Prerequisites for SQL Server 2016

- Two connected Azure Stack Hub integrated systems (Azure Stack Hub). This deployment doesn't work on the

Azure Stack Development Kit (ASDK). To learn more about Azure Stack Hub, see the Azure Stack overview.

- A tenant subscription on each Azure Stack Hub.
  - **Make a note of each subscription ID and the Azure Resource Manager endpoint for each Azure Stack Hub.**
- An Azure Active Directory (Azure AD) service principal that has permissions to the tenant subscription on each Azure Stack Hub. You may need to create two service principals if the Azure Stack Hubs are deployed against different Azure AD tenants. To learn how to create a service principal for Azure Stack Hub, see Create service principals to give apps access to Azure Stack Hub resources.
  - **Make a note of each service principal's application ID, client secret, and tenant name (xxxxx.onmicrosoft.com).**
- SQL Server 2016 Enterprise syndicated to each Azure Stack Hub's Marketplace. To learn more about marketplace syndication, see Download Marketplace items to Azure Stack Hub. **Make sure that your organization has the appropriate SQL licenses.**
- Docker for Windows installed on your local machine.

Get the Docker image

Docker images for each deployment eliminate dependency issues between different versions of Azure PowerShell.

1. Make sure that Docker for Windows is using Windows containers.

2. Run the following script in an elevated command prompt to get the Docker container with the deployment scripts.

```
docker pull intelligentedge/sqlserver2016-hadr:1.0.0
```

Deploy the availability group

1. Once the container image has been successfully pulled, start the image.

```
docker run -it intelligentedge/sqlserver2016-hadr:1.0.0 powershell
```

2. Once the container has started, you'll be given an elevated PowerShell terminal in the container. Change directories to get to the deployment script.

```
cd .\SQLHADRDemo\
```

3. Run the deployment. Provide credentials and resource names where needed. HA refers to the Azure Stack Hub where the HA cluster will be deployed. DR refers to the Azure Stack Hub where the DR cluster will be deployed.

```
> .\Deploy-AzureResourceGroup.ps1 `
-AzureStackApplicationId_HA "applicationIDforHAServicePrincipal" `
-AzureStackApplicationSercet_HA "clientSecretforHAServicePrincipal" `
-AADTenantName_HA "hatenantname.onmicrosoft.com" `
-AzureStackResourceGroup_HA "haresourcegroupname" `
-AzureStackArmEndpoint_HA "https://management.haazurestack.com" `
-AzureStackSubscriptionId_HA "haSubscriptionId" `
-AzureStackApplicationId_DR "applicationIDforDRServicePrincipal" `
-AzureStackApplicationSercet_DR "ClientSecretforDRServicePrincipal" `
-AADTenantName_DR "drtenantname.onmicrosoft.com" `
-AzureStackResourceGroup_DR "drresourcegroupname" `
-AzureStackArmEndpoint_DR "https://management.drazurestack.com" `
-AzureStackSubscriptionId_DR "drSubscriptionId"
```

4. Type `Y` to allow the NuGet provider to be installed, which will kick off the API Profile "2018-03-01-hybrid" modules to be installed.

5. Wait for resource deployment to complete.

6. Once DR resource deployment has completed, exit the container.

```
exit
```

7. Inspect the deployment by viewing the resources in each Azure Stack Hub's portal. Connect to one of the SQL instances on the HA environment and inspect the Availability Group through SQL Server Management Studio (SSMS).

Next steps

- Use SQL Server Management Studio to manually fail over the cluster. See Perform a Forced Manual Failover of an Always On Availability Group (SQL Server).
- Learn more about hybrid cloud apps. See Hybrid Cloud Solutions.
- Use your own data or modify the code to this sample on GitHub.

Direct traffic with a geo-distributed app using Azure and Azure Stack Hub

11/19/2021 • 18 minutes to read • Edit Online

Learn how to direct traffic to specific endpoints based on various metrics using the geo-distributed apps pattern. Creating a Traffic Manager profile with geographic-based routing and endpoint configuration ensures information is routed to endpoints based on regional requirements, corporate and international regulation, and your data needs.

In this solution, you'll build a sample environment to:

- Create a geo-distributed app.
- Use Traffic Manager to target your app.

Use the geo-distributed apps pattern

With the geo-distributed pattern, your app spans regions. You can default to the public cloud, but some of your users may require that their data remain in their region. You can direct users to the most suitable cloud based on their requirements.

Issues and considerations

Scalability considerations

The solution you'll build with this article isn't to accommodate scalability. However, if used in combination with other Azure and on-premises solutions, you can accommodate scalability requirements. For information on creating a hybrid solution with autoscaling via traffic manager, see Create cross-cloud scaling solutions with Azure.

Availability considerations

As is the case with scalability considerations, this solution doesn't directly address availability. However, Azure and on-premises solutions can be implemented within this solution to ensure high availability for all components involved.

When to use this pattern

- Your organization has international branches requiring custom regional security and distribution policies.
- Each of your organization's offices pulls employee, business, and facility data, which requires reporting activity per local regulations and time zones.
- High-scale requirements are met by horizontally scaling out apps with multiple app deployments within a single region and across regions to handle extreme load requirements.

Planning the topology

Before building out a distributed app footprint, it helps to know the following things:

- **Custom domain for the app:** What's the custom domain name that customers will use to access the app? For the sample app, the custom domain name is *www.scalableasedemo.com.*
- **Traffic Manager domain:** A domain name is chosen when creating an Azure Traffic Manager profile. This name is combined with the *trafficmanager.net* suffix to register a domain entry that's managed by Traffic Manager. For the sample app, the name chosen is *scalable-ase-demo*. As a result, the full domain name that's managed by Traffic Manager is *scalable-ase-demo.trafficmanager.net*.

- **Strategy for scaling the app footprint:** Decide whether the app footprint will be distributed across multiple App Service environments in a single region, multiple regions, or a mix of both approaches. The decision should be based on expectations of where customer traffic will originate and how well the rest of an app's supporting back-end infrastructure can scale. For example, with a 100% stateless app, an app can be massively scaled using a combination of multiple App Service environments per Azure region, multiplied by App Service environments deployed across multiple Azure regions. With 15+ global Azure regions available to choose from, customers can truly build a world-wide hyper-scale app footprint. For the sample app used here, three App Service environments were created in a single Azure region (South Central US).

- **Naming convention for the App Service environments:** Each App Service environment requires a unique name. Beyond one or two App Service environments, it's helpful to have a naming convention to help identify each App Service environment. For the sample app used here, a simple naming convention was used. The names of the three App Service environments are *fe1ase*, *fe2ase*, and *fe3ase*.

- **Naming convention for the apps:** Since multiple instances of the app will be deployed, a name is needed for each instance of the deployed app. With App Service Environment for Power Apps, the same app name can be used across multiple environments. Since each App Service environment has a unique domain suffix, developers can choose to reuse the exact same app name in each environment. For example, a developer could have apps named as follows: *myapp.foo1.p.azurewebsites.net*, *myapp.foo2.p.azurewebsites.net*, *myapp.foo3.p.azurewebsites.net*, and so on. For the app used here, each app instance has a unique name. The app instance names used are *webfrontend1*, *webfrontend2*, and *webfrontend3*.

TIP

Microsoft Azure Stack Hub is an extension of Azure. Azure Stack Hub brings the agility and innovation of cloud computing to your on-premises environment, enabling the only hybrid cloud that allows you to build and deploy hybrid apps anywhere.

The article Hybrid app design considerations reviews pillars of software quality (placement, scalability, availability, resiliency, manageability, and security) for designing, deploying, and operating hybrid apps. The design considerations assist in optimizing hybrid app design, minimizing challenges in production environments.

Part 1: Create a geo-distributed app

In this part, you'll create a web app.

- Create web apps and publish.
- Add code to Azure Repos.
- Point the app build to multiple cloud targets.
- Manage and configure the CD process.

Prerequisites

An Azure subscription and Azure Stack Hub installation are required.

Geo-distributed app steps

Obtain a custom domain and configure DNS

Update the DNS zone file for the domain. Azure AD can then verify ownership of the custom domain name. Use Azure DNS for Azure/Microsoft 365/external DNS records within Azure, or add the DNS entry at a different DNS

registrar.

1. Register a custom domain with a public registrar.
2. Sign in to the domain name registrar for the domain. An approved admin may be required to make the DNS updates.
3. Update the DNS zone file for the domain by adding the DNS entry provided by Azure AD. The DNS entry doesn't change behaviors such as mail routing or web hosting.

Create web apps and publish

Set up Hybrid Continuous Integration/Continuous Delivery (CI/CD) to deploy Web App to Azure and Azure Stack Hub, and auto push changes to both clouds.

> **NOTE**
>
> Azure Stack Hub with proper images syndicated to run (Windows Server and SQL) and App Service deployment are required. For more information, see Prerequisites for deploying App Service on Azure Stack Hub.

Add Code to Azure Repos

1. Sign in to Visual Studio with an **account that has project creation rights** on Azure Repos.

   CI/CD can apply to both app code and infrastructure code. Use Azure Resource Manager templates for both private and hosted cloud development.

2. **Clone the repository** by creating and opening the default web app.

Create web app deployment in both clouds

1. Edit the **WebApplication.csproj** file: Select `Runtimeidentifier` and add `win10-x64`. (See Self-contained Deployment documentation.)

2. **Check in the code to Azure Repos** using Team Explorer.

3. Confirm that the **application code** has been checked into Azure Repos.

Create the build definition

1. **Sign in to Azure Pipelines** to confirm ability to create build definitions.

2. Add `-r win10-x64` code. This addition is necessary to trigger a self-contained deployment with .NET Core.

3. **Run the build**. The self-contained deployment build process will publish artifacts that can run on Azure and Azure Stack Hub.

Using an Azure Hosted Agent

Using a hosted agent in Azure Pipelines is a convenient option to build and deploy web apps. Maintenance and upgrades are automatically performed by Microsoft Azure, which enables uninterrupted development, testing,

and deployment.

Manage and configure the CD process

Azure DevOps Services provide a highly configurable and manageable pipeline for releases to multiple environments such as development, staging, QA, and production environments; including requiring approvals at specific stages.

Create release definition

1. Select the **plus** button to add a new release under the **Releases** tab in the **Build and Release** section of Azure DevOps Services.

2. Apply the Azure App Service Deployment template.

3. Under **Add artifact**, add the artifact for the Azure Cloud build app.

4. Under Pipeline tab, select the **Phase, Task** link of the environment and set the Azure cloud environment values.

5. Set the **environment name** and select the **Azure subscription** for the Azure Cloud endpoint.

6. Under **App service name**, set the required Azure app service name.

7. Enter "Hosted VS2017" under **Agent queue** for Azure cloud hosted environment.

8. In Deploy Azure App Service menu, select the valid **Package or Folder** for the environment. Select **OK** to **folder location**.

9. Save all changes and go back to **release pipeline**.

10. Add a new artifact selecting the build for the Azure Stack Hub app.

11. Add one more environment by applying the Azure App Service Deployment.

12. Name the new environment Azure Stack Hub.

13. Find the Azure Stack Hub environment under **Task** tab.

14. Select the subscription for the Azure Stack Hub endpoint.

15. Set the Azure Stack Hub web app name as the App service name.

16. Select the Azure Stack Hub agent.

17. Under the Deploy Azure App Service section, select the valid **Package or Folder** for the environment. Select **OK** to folder location.

18. Under Variable tab add a variable named `VSTS\_ARM\_REST\_IGNORE\_SSL\_ERRORS`, set its value as **true**, and scope to Azure Stack Hub.

19. Select the **Continuous** deployment trigger icon in both artifacts and enable the **Continues** deployment trigger.

20. Select the **Pre-deployment** conditions icon in the Azure Stack Hub environment and set the trigger to **After release**.

21. Save all changes.

> **NOTE**
>
> Some settings for the tasks may have been automatically defined as environment variables when creating a release definition from a template. These settings can't be modified in the task settings; instead, the parent environment item must be selected to edit these settings.

Part 2: Update web app options

Azure App Service provides a highly scalable, self-patching web hosting service.

- Map an existing custom DNS name to Azure Web Apps.
- Use a **CNAME record** and an **A record** to map a custom DNS name to App Service.

Map an existing custom DNS name to Azure Web Apps

> **NOTE**
>
> Use a CNAME for all custom DNS names except a root domain (for example, northwind.com).

To migrate a live site and its DNS domain name to App Service, see Migrate an active DNS name to Azure App Service.

Prerequisites

To complete this solution:

- Create an App Service app, or use an app created for another solution.
- Purchase a domain name and ensure access to the DNS registry for the domain provider.

Update the DNS zone file for the domain. Azure AD will verify ownership of the custom domain name. Use Azure DNS for Azure/Microsoft 365/external DNS records within Azure, or add the DNS entry at a different DNS registrar.

- Register a custom domain with a public registrar.
- Sign in to the domain name registrar for the domain. (An approved admin may be required to make DNS updates.)
- Update the DNS zone file for the domain by adding the DNS entry provided by Azure AD.

For example, to add DNS entries for northwindcloud.com and www.northwindcloud.com, configure DNS settings for the northwindcloud.com root domain.

> **NOTE**
>
> A domain name may be purchased using the Azure portal. To map a custom DNS name to a web app, the web app's App Service plan must be a paid tier (**Shared**, **Basic**, **Standard**, or **Premium**).

Create and map CNAME and A records

Access DNS records with domain provider

> **NOTE**
>
> Use Azure DNS to configure a custom DNS name for Azure Web Apps. For more information, see Use Azure DNS to provide custom domain settings for an Azure service.

1. Sign in to the website of the main provider.
2. Find the page for managing DNS records. Every domain provider has its own DNS records interface. Look for areas of the site labeled **Domain Name**, **DNS**, or **Name Server Management**.

DNS records page can be viewed in **My domains**. Find the link named **Zone file**, **DNS Records**, or **Advanced configuration**.

The following screenshot is an example of a DNS records page:

1. In Domain Name Registrar, select **Add or Create** to create a record. Some providers have different links to add different record types. Consult the provider's documentation.
2. Add a CNAME record to map a subdomain to the app's default hostname.

For the www.northwindcloud.com domain example, add a CNAME record that maps the name to `<app_name>.azurewebsites.net`.

After adding the CNAME, the DNS records page looks like the following example:

Enable the CNAME record mapping in Azure

1. In a new tab, sign in to the Azure portal.
2. Go to App Services.
3. Select web app.
4. In the left navigation of the app page in the Azure portal, select **Custom domains**.
5. Select the **+** icon next to **Add hostname**.
6. Type the fully qualified domain name, like `www.northwindcloud.com`.
7. Select **Validate**.
8. If indicated, add additional records of other types (`A` or `TXT`) to the domain name registrars DNS records. Azure will provide the values and types of these records:

   a. An **A** record to map to the app's IP address.

   b. A **TXT** record to map to the app's default hostname `<app_name>.azurewebsites.net`. App Service uses this record only at configuration time to verify custom domain ownership. After verification, delete the TXT record.
9. Complete this task in the domain registrar tab and revalidate until the **Add hostname** button is activated.
10. Make sure that **Hostname record type** is set to **CNAME** (www.example.com or any subdomain).
11. Select **Add hostname**.
12. Type the fully qualified domain name, like `northwindcloud.com`.
13. Select **Validate**. The **Add** is activated.
14. Make sure that **Hostname record type** is set to **A record** (example.com).
15. **Add hostname**.

    It might take some time for the new hostnames to be reflected in the app's **Custom domains** page. Try refreshing the browser to update the data.

If there's an error, a verification error notification will appear at the bottom of the page.

> **NOTE**
> The above steps may be repeated to map a wildcard domain (*.northwindcloud.com). This allows the addition of any additional subdomains to this app service without having to create a separate CNAME record for each one. Follow the registrar instructions to configure this setting.

Test in a browser

Browse to the DNS name(s) configured earlier (for example, `northwindcloud.com` or `www.northwindcloud.com`).

Part 3: Bind a custom SSL cert

In this part, we will:

- Bind the custom SSL certificate to App Service.
- Enforce HTTPS for the app.
- Automate SSL certificate binding with scripts.

> **NOTE**
> If needed, obtain a customer SSL certificate in the Azure portal and bind it to the web app. For more information, see the App Service Certificates tutorial.

Prerequisites

To complete this solution:

- Create an App Service app.
- Map a custom DNS name to your web app.
- Acquire an SSL certificate from a trusted certificate authority and use the key to sign the request.

Requirements for your SSL certificate

To use a certificate in App Service, the certificate must meet all the following requirements:

- Signed by a trusted certificate authority.
- Exported as a password-protected PFX file.
- Contains private key at least 2048 bits long.
- Contains all intermediate certificates in the certificate chain.

> **NOTE**
>
> **Elliptic Curve Cryptography (ECC) certificates** work with App Service but aren't included in this guide. Consult a certificate authority for assistance in creating ECC certificates.

Prepare the web app

To bind a custom SSL certificate to the web app, the App Service plan must be in the **Basic**, **Standard**, or **Premium** tier.

Sign in to Azure

1. Open the Azure portal and go to the web app.

2. From the left menu, select **App Services**, and then select the web app name.

Check the pricing tier

1. In the left-hand navigation of the web app page, scroll to the **Settings** section and select **Scale up (App Service plan)**.

2. Ensure the web app isn't in the **Free** or **Shared** tier. The web app's current tier is highlighted in a dark blue box.

Custom SSL isn't supported in the **Free** or **Shared** tier. To upscale, follow the steps in the next section or the **Choose your pricing tier** page and skip to Upload and bind your SSL certificate.

Scale up your App Service plan

1. Select one of the **Basic**, **Standard**, or **Premium** tiers.

2. Select **Select**.

The scale operation is complete when notification is displayed.

Bind your SSL certificate and merge intermediate certificates

Merge multiple certificates in the chain.

1. **Open each certificate** you received in a text editor.

2. Create a file for the merged certificate called *mergedcertificate.crt*. In a text editor, copy the content of each certificate into this file. The order of your certificates should follow the order in the certificate chain, beginning with your certificate and ending with the root certificate. It looks like the following example:

```
-----BEGIN CERTIFICATE-----

<your entire Base64 encoded SSL certificate>

-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----

<The entire Base64 encoded intermediate certificate 1>

-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----

<The entire Base64 encoded intermediate certificate 2>

-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----

<The entire Base64 encoded root certificate>

-----END CERTIFICATE-----
```

Export certificate to PFX

Export the merged SSL certificate with the private key generated by the certificate.

A private key file is created via OpenSSL. To export the certificate to PFX, run the following command and replace the placeholders `<private-key-file>` and `<merged-certificate-file>` with the private key path and the merged certificate file:

```
openssl pkcs12 -export -out myserver.pfx -inkey <private-key-file> -in <merged-certificate-file>
```

When prompted, define an export password for uploading your SSL certificate to App Service later.

When IIS or **Certreq.exe** are used to generate the certificate request, install the certificate to a local machine and then export the certificate to PFX.

Upload the SSL certificate

1. Select **SSL settings** in the left navigation of the web app.
2. Select **Upload Certificate**.
3. In **PFX Certificate File**, select PFX file.
4. In **Certificate password**, type the password created when exporting the PFX file.
5. Select **Upload**.

When App Service finishes uploading the certificate, it appears in the **SSL settings** page.

Bind your SSL certificate

1. In the **SSL bindings** section, select **Add binding**.

> **NOTE**
>
> If the certificate has been uploaded, but doesn't appear in domain name(s) in the **Hostname** dropdown, try refreshing the browser page.

2. In the **Add SSL Binding** page, use the drop downs to select the domain name to secure and the certificate to use.

3. In **SSL Type**, select whether to use **Server Name Indication (SNI)** or IP-based SSL.

   - **SNI-based SSL**: Multiple SNI-based SSL bindings may be added. This option allows multiple SSL certificates to secure multiple domains on the same IP address. Most modern browsers (including Internet Explorer, Chrome, Firefox, and Opera) support SNI (find more comprehensive browser support information at Server Name Indication).

   - **IP-based SSL**: Only one IP-based SSL binding may be added. This option allows only one SSL certificate to secure a dedicated public IP address. To secure multiple domains, secure them all using the same SSL certificate. IP-based SSL is the traditional option for SSL binding.

4. Select **Add Binding**.

When App Service finishes uploading the certificate, it appears in the **SSL bindings** sections.

Remap the A record for IP SSL

If IP-based SSL isn't used in the web app, skip to Test HTTPS for your custom domain.

By default, the web app uses a shared public IP address. When the certificate is bound with IP-based SSL, App Service creates a new and dedicated IP address for the web app.

When an A record is mapped to the web app, the domain registry must be updated with the dedicated IP address.

The **Custom domain** page is updated with the new, dedicated IP address. Copy this IP address, then remap the A record to this new IP address.

Test HTTPS

In different browsers, go to `https://<your.custom.domain>` to ensure the web app is served.

> **NOTE**
> If certificate validation errors occur, a self-signed certificate may be the cause, or intermediate certificates may have been left off when exporting to the PFX file.

Enforce HTTPS

By default, anyone can access the web app using HTTP. All HTTP requests to the HTTPS port may be redirected.

In the web app page, select **SL settings**. Then, in **HTTPS Only**, select **On**.

When the operation is complete, go to any of the HTTP URLs that point to the app. For example:

- https://<app_name>.azurewebsites.net
- `https://northwindcloud.com`
- `https://www.northwindcloud.com`

Enforce TLS 1.1/1.2

The app allows TLS 1.0 by default, which is no longer considered secure by industry standards (like PCI DSS). To enforce higher TLS versions, follow these steps:

1. In the web app page, in the left navigation, select **SSL settings**.
2. In **TLS version**, select the minimum TLS version.

Create a Traffic Manager profile

1. Select **Create a resource** > **Networking** > **Traffic Manager profile** > **Create**.

2. In the **Create Traffic Manager profile**, complete as follows:

   a. In **Name**, provide a name for the profile. This name needs to be unique within the traffic manager.net zone and results in the DNS name, trafficmanager.net, which is used to access the Traffic Manager profile.

   b. In **Routing method**, select the **Geographic routing method**.

   c. In **Subscription**, select the subscription under which to create this profile.

   d. In **Resource Group**, create a new resource group to place this profile under.

   e. In **Resource group location**, select the location of the resource group. This setting refers to the location of the resource group and has no impact on the Traffic Manager profile deployed globally.

   f. Select **Create**.

   g. When the global deployment of the Traffic Manager profile is complete, it's listed in the respective resource group as one of the resources.

Add Traffic Manager endpoints

1. In the portal search bar, search for the **Traffic Manager profile** name created in the preceding section and select the traffic manager profile in the displayed results.

2. In **Traffic Manager profile**, in the **Settings** section, select **Endpoints**.

3. Select **Add**.

4. Adding the Azure Stack Hub Endpoint.

5. For **Type**, select **External endpoint**.

6. Provide a **Name** for this endpoint, ideally the name of the Azure Stack Hub.

7. For fully qualified domain name (**FQDN**), use the external URL for the Azure Stack Hub Web App.

8. Under Geo-mapping, select a region/continent where the resource is located. For example, **Europe.**

9. Under the Country/Region drop-down that appears, select the country that applies to this endpoint. For

example, **Germany**.

10. Keep **Add as disabled** unchecked.
11. Select **OK**.
12. Adding the Azure Endpoint:
    a. For **Type**, select **Azure endpoint**.
    b. Provide a **Name** for the endpoint.
    c. For **Target resource type**, select **App Service**.
    d. For **Target resource**, select **Choose an app service** to show the listing of the Web Apps under the same subscription. In **Resource**, pick the App service used as the first endpoint.
13. Under Geo-mapping, select a region/continent where the resource is located. For example, **North America/Central America/Caribbean.**
14. Under the Country/Region drop-down that appears, leave this spot blank to select all of the above regional grouping.
15. Keep **Add as disabled** unchecked.
16. Select **OK**.

> **NOTE**
> Create at least one endpoint with a geographic scope of All (World) to serve as the default endpoint for the resource.

17. When the addition of both endpoints is complete, they're displayed in **Traffic Manager profile** along with their monitoring status as **Online**.

Global Enterprise relies on Azure geo-distribution capabilities

Directing data traffic via Azure Traffic Manager and geography-specific endpoints enables global enterprises to adhere to regional regulations and keep data compliant and secure, which is crucial to the success of local and remote business locations.

Next steps

- To learn more about Azure Cloud Patterns, see Cloud Design Patterns.

Resilient identity and access management with Azure AD

11/19/2021 • 2 minutes to read • Edit Online

Identity and access management (IAM) is the process, policy, and technology framework that covers management of identities and what they can access. IAM includes components that support authentication and authorization of user and other accounts in a system.

Any component of an IAM system can cause disruption. IAM resilience is the ability to endure disruption to IAM system components and recover with minimal impact to business, users, customers, and operations. This guide describes ways to build a resilient IAM system.

To promote IAM resilience:

- Assume disruptions will occur, and plan for them.
- Reduce dependencies, complexity, and single points of failure.
- Ensure comprehensive error handling.

Recognizing and planning for contingencies is important. However, adding more identity systems, with their dependencies and complexity, could reduce rather than increase resilience.

Developers can help manage IAM resilience in their applications by using Azure AD Managed Identities wherever possible. For more information, see Increase resilience of authentication and authorization applications you develop.

When planning for resilience of an IAM solution, consider the following elements:

- The applications that rely on your IAM system.
- The public infrastructures that your authentication calls use, including:
  - Telecom companies.
  - Internet service providers.
  - Public key providers.
- Your cloud and on-premises identity providers.
- Other services that rely on your IAM, and APIs that connect the services.
- Any other on-premises components in your system.

Architecture

This diagram shows several ways to increase IAM resilience. The linked articles explain the methods in detail.

Manage dependencies and reduce authentication calls

Every authentication call is subject to disruption if any component of the call fails. When authentication is disrupted because of underlying component failures, users can't access their applications. So, reducing the number of authentication calls and the number of dependencies in those calls is essential for resilience.

- Manage dependencies. Build resilience with credential management.
- Reduce authentication calls. Build resilience with device states.
- Reduce external API dependencies.

Use long-lived revocable tokens

In a token-based authentication system like Azure AD, a user's client application must acquire a security token from the identity system before it can access an application or other resource. During the token validity period, the client can present the same token multiple times to access the application.

If the validity period expires during the user's session, the application rejects the token, and the client must acquire a new token from Azure AD. Acquiring a new token potentially requires user interaction like credential prompts or other requirements. Reducing the authentication call frequency with longer-lived tokens decreases unnecessary interactions. However, you must balance token life with the risk created by fewer policy evaluations.

- Use long-lived revocable tokens.
- Build resilience by using Continuous Access Evaluation (CAE).

For more information on managing token lifetimes, see Optimize reauthentication prompts and understand session lifetime for Azure AD Multi-Factor Authentication.

Hybrid and on-premises resilience

- Build resilience in your hybrid architecture to define resilient authentication from on-premises Active Directory or other identity providers (IdPs).
- To manage External Identities, build resilience in external user authentication.
- For accessing on-premises apps, build resilience in application access with Application Proxy.

Next steps

- Increase resilience of authentication and authorization applications you develop
- Build resilience in your IAM infrastructure

- Build resilience in your customer facing applications (CIAM) systems with Azure Active Directory B2C

Related resources

- Integrate on-premises AD domains with Azure AD
- Hybrid identity
- Manage identity in multitenant applications

Identity management in multitenant applications

11/19/2021 • 3 minutes to read • Edit Online

This series of articles describes best practices for multitenancy, when using Azure AD for authentication and identity management.

Sample code

When you're building a multitenant application, one of the first challenges is managing user identities, because now every user belongs to a tenant. For example:

- Users sign in with their organizational credentials.
- Users should have access to their organization's data, but not data that belongs to other tenants.
- An organization can sign up for the application, and then assign application roles to its members.

Azure Active Directory (Azure AD) has some great features that support all of these scenarios.

To accompany this series of articles, we created a complete end-to-end implementation of a multitenant application. The articles reflect what we learned in the process of building the application. To get started with the application, see the GitHub readme.

Introduction

Let's say you're writing an enterprise SaaS application to be hosted in the cloud. Of course, the application will have users:

But those users belong to organizations:

Example: Tailspin sells subscriptions to its SaaS application. Contoso and Fabrikam sign up for the app. When Alice (`alice@contoso`) signs in, the application should know that Alice is part of Contoso.

- Alice *should* have access to Contoso data.
- Alice *should not* have access to Fabrikam data.

This guidance will show you how to manage user identities in a multitenant application, using Azure Active Directory (Azure AD) to handle sign-in and authentication.

What is multitenancy?

A *tenant* is a group of users. In a SaaS application, the tenant is a subscriber or customer of the application. *Multitenancy* is an architecture where multiple tenants share the same physical instance of the app. Although tenants share physical resources (such as VMs or storage), each tenant gets its own logical instance of the app.

Typically, application data is shared among the users within a tenant, but not with other tenants.

Compare this architecture with a single-tenant architecture, where each tenant has a dedicated physical instance. In a single-tenant architecture, you add tenants by spinning up new instances of the app.

Multitenancy and horizontal scaling

To achieve scale in the cloud, it's common to add more physical instances. This is known as *horizontal scaling* or *scaling out*. Consider a web app. To handle more traffic, you can add more server VMs and put them behind a load balancer. Each VM runs a separate physical instance of the web app.

Any request can be routed to any instance. Together, the system functions as a single logical instance. You can tear down a VM or spin up a new VM, without affecting users. In this architecture, each physical instance is multitenant, and you scale by adding more instances. If one instance goes down, it should not affect any tenant.

Identity in a multitenant app

In a multitenant app, you must consider users in the context of tenants.

Authentication

- Users sign into the app with their organization credentials. They don't have to create new user profiles for the app.
- Users within the same organization are part of the same tenant.
- When a user signs in, the application knows which tenant the user belongs to.

Authorization

- When authorizing a user's actions (say, viewing a resource), the app must take into account the user's tenant.
- Users might be assigned roles within the application, such as "Admin" or "Standard User". Role assignments should be managed by the customer, not by the SaaS provider.

Example. Alice, an employee at Contoso, navigates to the application in her browser and clicks the "Log in" button. She is redirected to a sign-in screen where she enters her corporate credentials (username and password). At this point, she is logged into the app as `alice@contoso.com`. The application also knows that Alice is an admin user for this application. Because she is an admin, she can see a list of all the resources that belong to Contoso. However, she cannot view Fabrikam's resources, because she is an admin only within her tenant.

In this guidance, we'll look specifically at using Azure AD for identity management.

- We assume the customer stores their user profiles in Azure AD (including Office365 and Dynamics CRM tenants)
- Customers with on-premises Active Directory can use Azure AD Connect to sync their on-premises Active Directory with Azure AD. If a customer with on-premises Active Directory cannot use Azure AD Connect (due to corporate IT policy or other reasons), the SaaS provider can federate with the customer's directory through Active Directory Federation Services (AD FS). This option is described in Federating with a customer's AD FS.

This guidance does not consider other aspects of multitenancy such as data partitioning, per-tenant configuration, and so forth.

Next

The Tailspin scenario

11/19/2021 • 2 minutes to read • Edit Online

Sample code

Tailspin is a fictional company that is developing a SaaS application named Surveys. This application enables organizations to create and publish online surveys.

- An organization can sign up for the application.
- After the organization is signed up, users can sign into the application with their organizational credentials.
- Users can create, edit, and publish surveys.

> **NOTE**
>
> To get started with the application, see the GitHub readme.

Users can create, edit, and view surveys

An authenticated user can view all the surveys that he or she has created or has contributor rights to, and create new surveys. Notice that the user is signed in with his organizational identity, `bob@contoso.com`.

This screenshot shows the Edit Survey page:

Users can also view any surveys created by other users within the same tenant.

Survey owners can invite contributors

When a user creates a survey, he or she can invite other people to be contributors on the survey. Contributors can edit the survey, but cannot delete or publish it.

A user can add contributors from other tenants, which enables cross-tenant sharing of resources. In this screenshot, Bob (`bob@contoso.com`) is adding Alice (`alice@fabrikam.com`) as a contributor to a survey that Bob created.

When Alice logs in, she sees the survey listed under "Surveys I can contribute to".

Note that Alice signs into her own tenant, not as a guest of the Contoso tenant. Alice has contributor permissions only for that survey — she cannot view other surveys from the Contoso tenant.

Architecture

The Surveys application consists of a web front end and a web API backend. Both are implemented using ASP.NET Core.

The web application uses Azure Active Directory (Azure AD) to authenticate users. The web application also calls Azure AD to get OAuth 2 access tokens for the Web API. Access tokens are cached in Azure Cache for Redis. The cache enables multiple instances to share the same token cache (for example, in a server farm).

The diagram shows components in boxes, interacting with other components via two-way arrows. The Surveys web application authenticates with Azure AD to get access tokens for the web API, and caches the tokens in the Azure Cache for Redis access token cache.

Next

Authenticate using Azure AD and OpenID Connect

11/19/2021 • 7 minutes to read • Edit Online

Sample code

The Surveys application uses the OpenID Connect (OIDC) protocol to authenticate users with Azure Active Directory (Azure AD). The Surveys application uses ASP.NET Core, which has built-in middleware for OIDC. The following diagram shows what happens when the user signs in, at a high level.

1. The user clicks the "sign in" button in the app. This action is handled by an MVC controller.
2. The MVC controller returns a **ChallengeResult** action.
3. The middleware intercepts the **ChallengeResult** and creates a 302 response, which redirects the user to the Azure AD sign-in page.
4. The user authenticates with Azure AD.
5. Azure AD sends an ID token to the application.
6. The middleware validates the ID token. At this point, the user is now authenticated inside the application.
7. The middleware redirects the user back to application.

Register the app with Azure AD

To enable OpenID Connect, the SaaS provider registers the application inside their own Azure AD tenant.

To register the application, follow the steps in Quickstart: Register an application with the Microsoft identity platform.

To enable this functionality in the sample Surveys application, see the GitHub readme. Note the following:

- For a multitenant application, you must configure the multitenanted option explicitly. This enables other organizations to access the application.
- The reply URL is the URL where Azure AD will send OAuth 2.0 responses. When using the ASP.NET Core, this needs to match the path that you configure in the authentication middleware (see next section).

Configure the auth middleware

This section describes how to configure the authentication middleware in ASP.NET Core for multitenant authentication with OpenID Connect.

In your startup class, add the OpenID Connect middleware:

```
services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
  .AddMicrosoftIdentityWebApp(
    options =>
    {
        Configuration.Bind("AzureAd", options);
        options.Events = new SurveyAuthenticationEvents(loggerFactory);
        options.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        options.Events.OnTokenValidated += options.Events.TokenValidated;
    })
    .EnableTokenAcquisitionToCallDownstreamApi()
    .AddDownstreamWebApi(configOptions.SurveyApi.Name, Configuration.GetSection("SurveyApi"))
    .AddDistributedTokenCaches();
```

Notice that some of the settings are provided in the secrets.json file. The file must have a section named **AzureAd** with the following settings:

- **Instance**. For a multitenant application, set this to `https://login.microsoftonline.com`. This is the URL for the Azure AD common endpoint, which enables users from any Azure AD tenant to sign in.
- **ClientId**. The application's client ID, which you got when you registered the application in Azure AD.
- **TenantId**. GUID to sign in users in your organization.

Here's what the other middleware options mean:

- **SignInScheme**. Set this to `CookieAuthenticationDefaults.AuthenticationScheme`. This setting means that after the user is authenticated, the user claims are stored locally in a cookie. This cookie is how the user stays logged in during the browser session.
- **Events.** Event callbacks; see Authentication events.

Initiate the authentication flow

To start the authentication flow in ASP.NET MVC, return a **ChallengeResult** from the controller:

```
[AllowAnonymous]
public IActionResult SignIn()
{
    return new ChallengeResult(
        OpenIdConnectDefaults.AuthenticationScheme,
        new AuthenticationProperties
        {
            IsPersistent = true,
            RedirectUri = Url.Action("SignInCallback", "Account")
        });
}
```

This causes the middleware to return a 302 (Found) response that redirects to the authentication endpoint.

User login sessions

As mentioned, when the user first signs in, the Cookie Authentication middleware writes the user claims to a cookie. After that, HTTP requests are authenticated by reading the cookie.

By default, the cookie middleware writes a session cookie, which gets deleted once the user closes the browser. The next time the user next visits the site, they will have to sign in again. However, if you set **IsPersistent** to true in the **ChallengeResult**, the middleware writes a persistent cookie, so the user stays logged in after closing the browser. You can configure the cookie expiration; see Controlling cookie options. Persistent cookies are more convenient for the user, but may be inappropriate for some applications (say, a banking application) where you

want the user to sign in every time.

About the OpenID Connect middleware

The OpenID Connect middleware in ASP.NET hides most of the protocol details. This section contains some notes about the implementation, that may be useful for understanding the protocol flow.

First, let's examine the authentication flow in terms of ASP.NET (ignoring the details of the OIDC protocol flow between the app and Azure AD). The following diagram shows the process.

In this diagram, there are two MVC controllers. The Account controller handles sign-in requests, and the Home controller serves up the home page.

Here is the authentication process:

1. The user clicks the "Sign in" button, and the browser sends a GET request. For example: `GET /Account/SignIn/` .
2. The account controller returns a `ChallengeResult`.
3. The OIDC middleware returns an HTTP 302 response, redirecting to Azure AD.
4. The browser sends the authentication request to Azure AD
5. The user signs in to Azure AD, and Azure AD sends back an authentication response.
6. The OIDC middleware creates a claims principal and passes it to the Cookie Authentication middleware.
7. The cookie middleware serializes the claims principal and sets a cookie.
8. The OIDC middleware redirects to the application's callback URL.
9. The browser follows the redirect, sending the cookie in the request.
10. The cookie middleware deserializes the cookie to a claims principal and sets `HttpContext.User` equal to the claims principal. The request is routed to an MVC controller.

Authentication ticket

If authentication succeeds, the OIDC middleware creates an authentication ticket, which contains a claims principal that holds the user's claims.

> **NOTE**
> Until the entire authentication flow is completed, `HttpContext.User` still holds an anonymous principal, **not** the authenticated user. The anonymous principal has an empty claims collection. After authentication completes and the app redirects, the cookie middleware deserializes the authentication cookie and sets `HttpContext.User` to a claims principal that represents the authenticated user.

Authentication events

During the authentication process, the OpenID Connect middleware raises a series of events:

- **RedirectToIdentityProvider**. Called right before the middleware redirects to the authentication endpoint. You can use this event to modify the redirect URL; for example, to add request parameters. See Adding the admin consent prompt for an example.
- **AuthorizationCodeReceived**. Called with the authorization code.
- **TokenResponseReceived**. Called after the middleware gets an access token from the IDP, but before it is validated. Applies only to authorization code flow.
- **TokenValidated**. Called after the middleware validates the ID token. At this point, the application has a set of validated claims about the user. You can use this event to perform additional validation on the claims, or to transform claims. See Working with claims.
- **UserInformationReceived**. Called if the middleware gets the user profile from the user info endpoint. Applies only to authorization code flow, and only when `GetClaimsFromUserInfoEndpoint = true` in the middleware options.
- **TicketReceived**. Called when authentication is completed. This is the last event, assuming that authentication succeeds. After this event is handled, the user is signed into the app.
- **AuthenticationFailed**. Called if authentication fails. Use this event to handle authentication failures — for example, by redirecting to an error page.

To provide callbacks for these events, set the **Events** option on the middleware. There are two different ways to declare the event handlers: Inline with lambdas, or in a class that derives from **OpenIdConnectEvents**. The second approach is recommended if your event callbacks have any substantial logic, so they don't clutter your startup class. Our reference implementation uses this approach.

OpenID Connect endpoints

Azure AD supports OpenID Connect Discovery, wherein the identity provider (IDP) returns a JSON metadata document from a well-known endpoint. The metadata document contains information such as:

- The URL of the authorization endpoint. This is where the app redirects to authenticate the user.
- The URL of the "end session" endpoint, where the app goes to log out the user.
- The URL to get the signing keys, which the client uses to validate the OIDC tokens that it gets from the IDP.

By default, the OIDC middleware knows how to fetch this metadata. Set the **Authority** option in the middleware, and the middleware constructs the URL for the metadata. (You can override the metadata URL by setting the **MetadataAddress** option.)

OpenID Connect flows

By default, the OIDC middleware uses hybrid flow with form post response mode.

- *Hybrid flow* means the client can get an ID token and an authorization code in the same round-trip to the authorization server.
- *Form post response mode* means the authorization server uses an HTTP POST request to send the ID token and authorization code to the app. The values are form-urlencoded (content type = "application/x-www-form-urlencoded").

When the OIDC middleware redirects to the authorization endpoint, the redirect URL includes all of the query string parameters needed by OIDC. For hybrid flow:

- client_id. This value is set in the **ClientId** option.
- scope = "openid profile", which means it's an OIDC request and we want the user's profile.
- response_type = "code id_token". This specifies hybrid flow.
- response_mode = "form_post". This specifies form post response.

To specify a different flow, set the **ResponseType** property on the options.

```
app.AddAuthentication().AddOpenIdConnect(options =>
{
    options.ResponseType = "code"; // Authorization code flow

    // Other options
}
```

Next

Work with claims-based identities

11/19/2021 • 5 minutes to read • Edit Online

Sample code

Claims in Azure AD

When a user signs in, Azure AD sends an ID token that contains a set of claims about the user. A claim is simply a piece of information, expressed as a key/value pair. For example, `email` = `bob@contoso.com`. Claims have an issuer (in this case, Azure AD), which is the entity that authenticates the user and creates the claims. You trust the claims because you trust the issuer. (Conversely, if you don't trust the issuer, don't trust the claims!)

At a high level:

1. The user authenticates.
2. The Identity Provider (IDP) sends a set of claims.
3. The app normalizes or augments the claims (optional).
4. The app uses the claims to make authorization decisions.

In OpenID Connect, the set of claims that you get is controlled by the scope parameter of the authentication request. However, Azure AD issues a limited set of claims through OpenID Connect; see Supported Token and Claim Types. If you want more information about the user, you'll need to use the Azure AD Graph API.

Here are some of the claims from Azure AD that an app might typically care about:

CLAIM TYPE IN ID TOKEN	DESCRIPTION
aud	Who the token was issued for. This will be the application's client ID. Generally, you shouldn't need to worry about this claim, because the middleware automatically validates it. Example: `"91464657-d17a-4327-91f3-2ed99386406f"`
groups	A list of Azure AD groups of which the user is a member. Example: `["93e8f556-8661-4955-87b6-890bc043c30f", "fc781505-18ef-4a31-a7d5-7d931d7b857e"]`
iss	The issuer of the OIDC token. Example: `https://sts.windows.net/b9bd2162-77ac-4fb2-8254-5c36e9c0a9c4/`
name	The user's display name. Example: `"Alice A."`
oid	The object identifier for the user in Azure AD. This value is the immutable and non-reusable identifier of the user. Use this value, not email, as a unique identifier for users; email addresses can change. If you use the Azure AD Graph API in your app, object ID is that value used to query profile information. Example: `"59f9d2dc-995a-4ddf-915e-b3bb314a7fa4"`

CLAIM TYPE IN ID TOKEN	DESCRIPTION
roles	A list of app roles for the user. Example: `["SurveyCreator"]`
tid	Tenant ID. This value is a unique identifier for the tenant in Azure AD. Example: `"b9bd2162-77ac-4fb2-8254-5c36e9c0a9c4"`
unique_name	A human readable display name of the user. Example: `"alice@contoso.com"`
upn	User principal name. Example: `"alice@contoso.com"`

This table lists the claim types as they appear in the ID token. In ASP.NET Core, the OpenID Connect middleware converts some of the claim types when it populates the Claims collection for the user principal:

- oid > `http://schemas.microsoft.com/identity/claims/objectidentifier`
- tid > `http://schemas.microsoft.com/identity/claims/tenantid`
- unique_name > `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name`
- upn > `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn`

Claims transformations

During the authentication flow, you might want to modify the claims that you get from the IDP. In ASP.NET Core, you can perform claims transformation inside of the **AuthenticationValidated** event from the OpenID Connect middleware. (See Authentication events.)

Any claims that you add during **AuthenticationValidated** are stored in the session authentication cookie. They don't get pushed back to Azure AD.

Here are some examples of claims transformation:

- **Claims normalization**, or making claims consistent across users. This is particularly relevant if you are getting claims from multiple IDPs, which might use different claim types for similar information.
- Add **default claim values** for claims that aren't present (for example, assigning a user to a default role). In some cases this can simplify authorization logic.
- Add **custom claim types** with application-specific information about the user. For example, you might store some information about the user in a database. You could add a custom claim with this information to the authentication ticket. The claim is stored in a cookie, so you only need to get it from the database once per login session. On the other hand, you also want to avoid creating excessively large cookies, so you need to consider the trade-off between cookie size versus database lookups.

After the authentication flow is complete, the claims are available in `HttpContext.User`. At that point, you should treat them as a read-only collection—for example, using them to make authorization decisions.

Issuer validation

In OpenID Connect, the issuer claim ("iss") identifies the IDP that issued the ID token. Part of the OIDC authentication flow is to verify that the issuer claim matches the actual issuer. The OIDC middleware handles this for you.

In Azure AD, the issuer value is unique per AD tenant (`https://sts.windows.net/<tenantID>`). Therefore, an application should do another check, to make sure the issuer represents a tenant that is allowed to sign in to the

app.

For a single-tenant application, you can just check that the issuer is your own tenant. In fact, the OIDC middleware does this automatically by default. In a multitenant app, you need to allow for multiple issuers, corresponding to the different tenants. Here is a general approach to use:

- When a tenant signs up, store the tenant and the issuer in your user DB.
- Whenever a user signs in, look up the issuer in the database. If the issuer isn't found, it means that tenant hasn't signed up. You can redirect them to a sign-up page.
- You could also block certain tenants; for example, for customers that didn't pay their subscription.

For a more detailed discussion, see Sign-up and tenant onboarding in a multitenant application.

Using claims for authorization

With claims, a user's identity is no longer a monolithic entity. For example, a user might have an email address, phone number, birthday, gender, etc. Maybe the user's IDP stores all of this information. But when you authenticate the user, you'll typically get a subset of these as claims. In this model, the user's identity is simply a bundle of claims. When you make authorization decisions about a user, you will look for particular sets of claims. In other words, the question "Can user X perform action Y" ultimately becomes "Does user X have claim Z".

Here are some basic patterns for checking claims.

- To check that the user has a particular claim with a particular value:

  ```
  if (User.HasClaim(ClaimTypes.Role, "Admin")) { ... }
  ```

  This code checks whether the user has a Role claim with the value "Admin". It correctly handles the case where the user has no Role claim or multiple Role claims.

  The **ClaimTypes** class defines constants for commonly used claim types. However, you can use any string value for the claim type.

- To get a single value for a claim type, when you expect there to be at most one value:

  ```
  string email = User.FindFirst(ClaimTypes.Email)?.Value;
  ```

- To get all the values for a claim type:

  ```
  IEnumerable<Claim> groups = User.FindAll("groups");
  ```

For more information, see Role-based and resource-based authorization in multitenant applications.

Next

Tenant sign-up and onboarding

11/19/2021 • 6 minutes to read • Edit Online

Sample code

This article describes how to implement a *sign-up* process in a multitenant application, which allows a customer to sign up their organization for your application.

There are several reasons to implement a sign-up process:

- Allow an AD admin to consent for the customer's entire organization to use the application.
- Collect credit card payment or other customer information.
- Perform any one-time per-tenant setup needed by your application.

Admin consent and Azure AD permissions

In order to authenticate with Azure AD, an application needs access to the user's directory. At a minimum, the application needs permission to read the user's profile. The first time that a user signs in, Azure AD shows a consent page that lists the permissions being requested. By clicking **Accept**, the user grants permission to the application.

By default, consent is granted on a per-user basis. Every user who signs in sees the consent page. However, Azure AD also supports *admin consent*, which allows an AD administrator to consent for an entire organization.

When the admin consent flow is used, the consent page states that the AD admin is granting permission on behalf of the entire tenant:

After the admin clicks **Accept**, other users within the same tenant can sign in, and Azure AD will skip the consent screen.

Only an AD administrator can give admin consent, because it grants permission on behalf of the entire organization. If a non-administrator tries to authenticate with the admin consent flow, Azure AD displays an error:

Additional technical information:
Correlation ID: 662b80ea-fb8b-4fe9-a368-02f597854249
Timestamp: 2015-11-12 23:28:46Z
AADSTS90093: This operation can only be performed by an administrator. Sign out and sign in as an administrator or contact one of your organization's administrators.

If the application requires additional permissions at a later point, the customer will need to sign up again and consent to the updated permissions.

Implementing tenant sign-up

For the Tailspin Surveys application, we defined several requirements for the sign-up process:

- A tenant must sign up before users can sign in.
- Sign-up uses the admin consent flow.
- Sign-up adds the user's tenant to the application database.
- After a tenant signs up, the application shows an onboarding page.

In this section, we'll walk through our implementation of the sign-up process. It's important to understand that "sign up" versus "sign in" is an application concept. During the authentication flow, Azure AD does not inherently know whether the user is in process of signing up. It's up to the application to keep track of the context.

When an anonymous user visits the Surveys application, the user is shown two buttons, one to sign in, and one to "enroll your company" (sign up).

These buttons invoke actions in the `AccountController` class.

The `SignIn` action returns a **ChallengeResult**, which causes the OpenID Connect middleware to redirect to the authentication endpoint. This is the default way to trigger authentication in ASP.NET Core.

```
[AllowAnonymous]
public IActionResult SignIn()
{
    return new ChallengeResult(
        OpenIdConnectDefaults.AuthenticationScheme,
        new AuthenticationProperties
        {
            IsPersistent = true,
            RedirectUri = Url.Action("SignInCallback", "Account")
        });
}
```

Now compare the `SignUp` action:

```
[AllowAnonymous]
public IActionResult SignUp()
{
    var state = new Dictionary<string, string> { { "signup", "true" }};
    return new ChallengeResult(
        OpenIdConnectDefaults.AuthenticationScheme,
        new AuthenticationProperties(state)
        {
            RedirectUri = Url.Action(nameof(SignUpCallback), "Account")
        });
}
```

Like `SignIn`, the `SignUp` action also returns a `ChallengeResult`. But this time, we add a piece of state information to the `AuthenticationProperties` in the `ChallengeResult`:

- signup: A Boolean flag, indicating that the user has started the sign-up process.

The state information in `AuthenticationProperties` gets added to the OpenID Connect state parameter, which round trips during the authentication flow.

After the user authenticates in Azure AD and gets redirected back to the application, the authentication ticket contains the state. We are using this fact to make sure the "signup" value persists across the entire authentication flow.

Adding the admin consent prompt

In Azure AD, the admin consent flow is triggered by adding a "prompt" parameter to the query string in the authentication request:

```
/authorize?prompt=admin_consent&...
```

The Surveys application adds the prompt during the `RedirectToIdentityProvider` event. This event is called right before the middleware redirects to the authentication endpoint.

```
public override Task RedirectToIdentityProvider(RedirectContext context)
{
    if (context.IsSigningUp())
    {
        context.ProtocolMessage.Prompt = "admin_consent";
    }

    _logger.RedirectToIdentityProvider();
    return Task.FromResult(0);
}
```

Setting `ProtocolMessage.Prompt` tells the middleware to add the "prompt" parameter to the authentication request.

Note that the prompt is only needed during sign-up. Regular sign-in should not include it. To distinguish between them, we check for the `signup` value in the authentication state. The following extension method checks for this condition:

```
internal static bool IsSigningUp(this AuthenticationProperties properties)
{
    Guard.ArgumentNotNull(properties, nameof(properties));

    string signupValue = string.Empty;
    // Check the HTTP context and convert to string
    if ((properties == null) ||
        (!properties.Items.TryGetValue("signup", out signupValue)))
    {
        return false;
    }

    // We have found the value, so see if it's valid
    bool isSigningUp;
    if (!bool.TryParse(signupValue, out isSigningUp))
    {
        // The value for signup is not a valid boolean, throw

        throw new InvalidOperationException($"'{signupValue}' is an invalid boolean value");
    }

    return isSigningUp;
}
```

Registering a tenant

The Surveys application stores some information about each tenant and user in the application database.

In the Tenant table, IssuerValue is the value of the issuer claim for the tenant. For Azure AD, this is `https://sts.windows.net/<tentantID>` and gives a unique value per tenant.

When a new tenant signs up, the Surveys application writes a tenant record to the database. This happens inside the `AuthenticationValidated` event. (Don't do it before this event, because the ID token won't be validated yet, so you can't trust the claim values. See Authentication.

Here is the relevant code from the Surveys application:

```
public override async Task TokenValidated(TokenValidatedContext context)
{
    var principal = context.Principal;
    var userId = principal.GetObjectIdentifierValue();
    var tenantManager = context.HttpContext.RequestServices.GetService<TenantManager>();
    var userManager = context.HttpContext.RequestServices.GetService<UserManager>();
    var issuerValue = context.SecurityToken.Issuer;
    _logger.AuthenticationValidated(userId, issuerValue);

    // Normalize the claims first.
    NormalizeClaims(principal);
    var tenant = await tenantManager.FindByIssuerValueAsync(issuerValue);

    if (context.Properties.IsSigningUp())
    {
        if (tenant == null)
        {
            tenant = await SignUpTenantAsync(context, tenantManager)
                .ConfigureAwait(false);
        }

        // In this case, we need to go ahead and set up the user signing us up.
        await CreateOrUpdateUserAsync(context.Ticket, userManager, tenant)
            .ConfigureAwait(false);
    }
    else
    {
        if (tenant == null)
        {
            _logger.UnregisteredUserSignInAttempted(userId, issuerValue);
            throw new SecurityTokenValidationException($"Tenant {issuerValue} is not registered");
        }

        await CreateOrUpdateUserAsync(context.Principal, userManager, tenant)
            .ConfigureAwait(false);
    }
}
```

This code does the following:

1. Check if the tenant's issuer value is already in the database. If the tenant has not signed up, `FindByIssuerValueAsync` returns null.
2. If the user is signing up:
   a. Add the tenant to the database (`SignUpTenantAsync`).
   b. Add the authenticated user to the database (`CreateOrUpdateUserAsync`).
3. Otherwise complete the normal sign-in flow:
   a. If the tenant's issuer was not found in the database, it means the tenant is not registered, and the customer needs to sign up. In that case, throw an exception to cause the authentication to fail.
   b. Otherwise, create a database record for this user, if there isn't one already (`CreateOrUpdateUserAsync`).

Here is the `SignUpTenantAsync` method that adds the tenant to the database.

```
private async Task<Tenant> SignUpTenantAsync(TokenValidatedContext context, TenantManager tenantManager)
{
    Guard.ArgumentNotNull(context, nameof(context));
    Guard.ArgumentNotNull(tenantManager, nameof(tenantManager));

    var principal = context.Principal;
    var issuerValue = principal.GetIssuerValue();
    var tenant = new Tenant
    {
        IssuerValue = issuerValue,
        Created = DateTimeOffset.UtcNow
    };

    try
    {
        await tenantManager.CreateAsync(tenant)
            .ConfigureAwait(false);
    }
    catch(Exception ex)
    {
        _logger.SignUpTenantFailed(principal.GetObjectIdentifierValue(), issuerValue, ex);
        throw;
    }

    return tenant;
}
```

Here is a summary of the entire sign-up flow in the Surveys application:

1. The user clicks the **Sign Up** button.
2. The `AccountController.SignUp` action returns a challenge result. The authentication state includes "signup" value.
3. In the `RedirectToAuthenticationEndpoint` event, add the `admin_consent` prompt.
4. The OpenID Connect middleware redirects to Azure AD and the user authenticates.
5. In the `AuthenticationValidated` event, look for the "signup" state.
6. Add the tenant to the database.

Next

Application roles

11/19/2021 • 5 minutes to read • Edit Online

Sample code

Application roles are used to assign permissions to users. For example, the Tailspin Surveys application defines the following roles:

- Administrator. Can perform all CRUD operations on any survey that belongs to that tenant.
- Creator. Can create new surveys.
- Reader. Can read any surveys that belong to that tenant.

You can see that roles ultimately get translated into permissions, during authorization. But the first question is how to assign and manage roles. We identified three main options:

- Azure AD App Roles
- Azure AD security groups
- Application role manager.

Roles using Azure AD App Roles

This is the approach that we used in the Tailspin Surveys app.

In this approach, The SaaS provider defines the application roles by adding them to the application manifest. After a customer signs up, an admin for the customer's AD directory assigns users to the roles. When a user signs in, the user's assigned roles are sent as claims.

> **NOTE**
>
> If the customer has Azure AD Premium, the admin can assign a security group to a role, and user members of the group will inherit the app role. This is a convenient way to manage roles, because the group owner doesn't need to be an admin or app owner.

Advantages of this approach:

- Simple programming model.
- Roles are specific to the application. The role claims for one application are not sent to another application.
- If the customer removes the application from their Azure AD tenant, the roles go away.
- The application doesn't need any extra Azure AD permissions, other than reading the user's profile.

Drawbacks:

- Customers without Azure AD Premium cannot assign app roles to security groups. For these customers, all app role assignments to users must be done individually, by an administrator or an owner of the app.
- If you have a backend web API which is separate from the web app, the app role assignments for the web app don't apply to the web API. For more discussion of this point, see Securing a backend web API.

Implementation

Define the roles. The SaaS provider declares the app roles in the application manifest. For example, here is the manifest entry for the Surveys app:

```
"appRoles": [
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Creators can create Surveys",
    "displayName": "SurveyCreator",
    "id": "1b4f816e-5eaf-48b9-8613-7923830595ad",
    "isEnabled": true,
    "value": "SurveyCreator"
  },
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Administrators can manage the Surveys in their tenant",
    "displayName": "SurveyAdmin",
    "id": "c20e145e-5459-4a6c-a074-b942bbd4cfe1",
    "isEnabled": true,
    "value": "SurveyAdmin"
  }
],
```

The `value` property appears in the role claim. The `id` property is the unique identifier for the defined role. Always generate a new GUID value for `id`.

Assign users. When a new customer signs up, the application is registered in the customer's Azure AD tenant. At this point, an Azure AD admin for that tenant or an app owner (under Enterprise apps) can assign app roles to users.

> **NOTE**
>
> As noted earlier, customers with Azure AD Premium can also assign app roles to security groups.

The following screenshot from the Azure portal shows users and groups for the Survey application. Admin and Creator are groups, assigned the SurveyAdmin and SurveyCreator app roles, respectively. Alice is a user who was assigned the SurveyAdmin app role directly. Bob and Charles are users that have not been directly assigned an app role.

DISPLAY NAME	OBJECT TYPE	ROLE ASSIGNED
Admin	Group	SurveyAdmin
Alice	User	SurveyAdmin
Bob	User	Default Access
Charles	User	Default Access
Creators	Group	SurveyCreator

As shown in the following screenshot, Charles is part of the Admin group, so he inherits the SurveyAdmin role. In the case of Bob, he has not been assigned an app role yet.

> **NOTE**
> An alternative approach is for the application to assign app roles programmatically, using the Azure AD Graph API. However, this requires the application to obtain write permissions for the customer's Azure AD directory, which is a high privilege that is usually unnecessary.

Get role claims. When a user signs in, the application receives the user's assigned role(s) in a claim with type `http://schemas.microsoft.com/ws/2008/06/identity/claims/role` (the `roles` claim in a JWT token).

A user can be assigned multiple roles, or no role. In your authorization code, don't assume the user has exactly one role claim. Instead, write code that checks whether a particular claim value is present:

```
if (context.User.HasClaim(ClaimTypes.Role, "Admin")) { ... }
```

Roles using Azure AD security groups

In this approach, roles are represented as Azure AD security groups. The application assigns permissions to users based on their security group memberships.

Advantages:

- For customers who do not have Azure AD Premium, this approach enables the customer to use security groups to manage role assignments.

Disadvantages:

- Complexity. Because every tenant sends different group claims, the app must keep track of which security groups correspond to which application roles, for each tenant.
- As users belong to more groups, access tokens grow to include more claims. After a certain limit, Azure AD includes an "overage" claim to limit the token size; see Microsoft identity platform access tokens. Application roles avoid this issue because they are scoped to the specific application.

Implementation

In the application manifest, set the `groupMembershipClaims` property to "SecurityGroup". This is needed to get group membership claims from Azure AD.

```
{
   // ...
   "groupMembershipClaims": "SecurityGroup",
}
```

When a new customer signs up, the application instructs the customer to create security groups for the roles

needed by the application. The customer then needs to enter the group object IDs into the application. The application stores these in a table that maps group IDs to application roles, per tenant.

> **NOTE**
> Alternatively, the application could create the groups programmatically, using the Microsoft Graph API. This could be less error prone, but requires the application to obtain privileged read/write permissions for the customer's directory. Many customers might be unwilling to grant this level of access.

When a user signs in:

1. The application receives the user's groups as claims. The value of each claim is the object ID of a group.
2. Azure AD limits the number of groups sent in the token. If the number of groups exceeds this limit, Azure AD sends a special "overage" claim. If that claim is present, the application must query the Azure AD Graph API to get all of the groups to which that user belongs. For details, see Groups overage claim.
3. The application looks up the object IDs in its own database, to find the corresponding application roles to assign to the user.
4. The application adds a custom claim value to the user principal that expresses the application role. For example: `survey_role` = "SurveyAdmin".

Authorization policies should use the custom role claim, not the group claim.

Roles using an application role manager

With this approach, application roles are not stored in Azure AD at all. Instead, the application stores the role assignments for each user in its own DB — for example, using the **RoleManager** class in ASP.NET Identity.

Advantages:

- The app has full control over the roles and user assignments.

Drawbacks:

- More complex, harder to maintain.
- Cannot use Azure AD security groups to manage role assignments.
- Stores user information in the application database, where it can get out of sync with the tenant's Azure AD directory, as users are added or removed.

Next

Role-based and resource-based authorization

11/19/2021 • 5 minutes to read • Edit Online

Sample code

Our reference implementation is an ASP.NET Core application. In this article we'll look at two general approaches to authorization, using the authorization APIs provided in ASP.NET Core.

- **Role-based authorization**. Authorizing an action based on the roles assigned to a user. For example, some actions require an administrator role.
- **Resource-based authorization**. Authorizing an action based on a particular resource. For example, every resource has an owner. The owner can delete the resource; other users cannot.

A typical app will employ a mix of both. For example, to delete a resource, the user must be the resource owner *or* an admin.

Role-based authorization

The Tailspin Surveys application defines the following roles:

- Administrator. Can perform all CRUD operations on any survey that belongs to that tenant.
- Creator. Can create new surveys
- Reader. Can read any surveys that belong to that tenant

Roles apply to *users* of the application. In the Surveys application, a user is either an administrator, creator, or reader.

For a discussion of how to define and manage roles, see Application roles.

Regardless of how you manage the roles, your authorization code will look similar. ASP.NET Core has an abstraction called authorization policies. With this feature, you define authorization policies in code, and then apply those policies to controller actions. The policy is decoupled from the controller.

Create policies

To define a policy, first create a class that implements `IAuthorizationRequirement`. It's easiest to derive from `AuthorizationHandler`. In the `Handle` method, examine the relevant claim(s).

Here is an example from the Tailspin Surveys application:

```
public class SurveyCreatorRequirement : AuthorizationHandler<SurveyCreatorRequirement>,
IAuthorizationRequirement
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
SurveyCreatorRequirement requirement)
    {
        if (context.User.HasClaim(ClaimTypes.Role, Roles.SurveyAdmin) ||
            context.User.HasClaim(ClaimTypes.Role, Roles.SurveyCreator))
        {
            context.Succeed(requirement);
        }
        return Task.FromResult(0);
    }
}
```

This class defines the requirement for a user to create a new survey. The user must be in the SurveyAdmin or SurveyCreator role.

In your startup class, define a named policy that includes one or more requirements. If there are multiple requirements, the user must meet *every* requirement to be authorized. The following code defines two policies:

```
services.AddAuthorization(options =>
{
    options.AddPolicy(PolicyNames.RequireSurveyCreator,
        policy =>
        {
            policy.AddRequirements(new SurveyCreatorRequirement());
            policy.RequireAuthenticatedUser(); // Adds DenyAnonymousAuthorizationRequirement
            // By adding the CookieAuthenticationDefaults.AuthenticationScheme, if an authenticated
            // user is not in the appropriate role, they will be redirected to a "forbidden" page.
            policy.AddAuthenticationSchemes(CookieAuthenticationDefaults.AuthenticationScheme);
        });

    options.AddPolicy(PolicyNames.RequireSurveyAdmin,
        policy =>
        {
            policy.AddRequirements(new SurveyAdminRequirement());
            policy.RequireAuthenticatedUser();
            policy.AddAuthenticationSchemes(CookieAuthenticationDefaults.AuthenticationScheme);
        });
});
```

This code also sets the authentication scheme, which tells ASP.NET which authentication middleware should run if authorization fails. In this case, we specify the cookie authentication middleware, because the cookie authentication middleware can redirect the user to a "Forbidden" page. The location of the Forbidden page is set in the `AccessDeniedPath` option for the cookie middleware; see Configuring the authentication middleware.

Authorize controller actions

Finally, to authorize an action in an MVC controller, set the policy in the `Authorize` attribute:

```
[Authorize(Policy = PolicyNames.RequireSurveyCreator)]
public IActionResult Create()
{
    var survey = new SurveyDTO();
    return View(survey);
}
```

In earlier versions of ASP.NET, you would set the **Roles** property on the attribute:

```
// old way
[Authorize(Roles = "SurveyCreator")]
```

This is still supported in ASP.NET Core, but it has some drawbacks compared with authorization policies:

- It assumes a particular claim type. Policies can check for any claim type. Roles are just a type of claim.
- The role name is hard-coded into the attribute. With policies, the authorization logic is all in one place, making it easier to update or even load from configuration settings.
- Policies enable more complex authorization decisions (for example, age >= 21) that can't be expressed by simple role membership.

Resource-based authorization

Resource based authorization occurs whenever the authorization depends on a specific resource that will be

affected by an operation. In the Tailspin Surveys application, every survey has an owner and zero-to-many contributors.

- The owner can read, update, delete, publish, and unpublish the survey.
- The owner can assign contributors to the survey.
- Contributors can read and update the survey.

Note that "owner" and "contributor" are not application roles; they are stored per survey, in the application database. To check whether a user can delete a survey, for example, the app checks whether the user is the owner for that survey.

In ASP.NET Core, implement resource-based authorization by deriving from **AuthorizationHandler** and overriding the **Handle** method.

```
public class SurveyAuthorizationHandler : AuthorizationHandler<OperationAuthorizationRequirement, Survey>
{
    protected override void HandleRequirementAsync(AuthorizationHandlerContext context,
OperationAuthorizationRequirement operation, Survey resource)
    {
    }
}
```

Notice that this class is strongly typed for Survey objects. Register the class for DI on startup:

```
services.AddSingleton<IAuthorizationHandler>(factory =>
{
    return new SurveyAuthorizationHandler();
});
```

To perform authorization checks, use the **IAuthorizationService** interface, which you can inject into your controllers. The following code checks whether a user can read a survey:

```
if (await _authorizationService.AuthorizeAsync(User, survey, Operations.Read) == false)
{
    return StatusCode(403);
}
```

Because we pass in a `Survey` object, this call will invoke the `SurveyAuthorizationHandler`.

In your authorization code, a good approach is to aggregate all of the user's role-based and resource-based permissions, then check the aggregate set against the desired operation. Here is an example from the Surveys app. The application defines several permission types:

- Admin
- Contributor
- Creator
- Owner
- Reader

The application also defines a set of possible operations on surveys:

- Create
- Read
- Update
- Delete

- Publish
- Unpublish

The following code creates a list of permissions for a particular user and survey. Notice that this code looks at both the user's app roles, and the owner/contributor fields in the survey.

```
public class SurveyAuthorizationHandler : AuthorizationHandler<OperationAuthorizationRequirement, Survey>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
OperationAuthorizationRequirement requirement, Survey resource)
    {
        var permissions = new List<UserPermissionType>();
        int surveyTenantId = context.User.GetSurveyTenantIdValue();
        int userId = context.User.GetSurveyUserIdValue();
        string user = context.User.GetUserName();

        if (resource.TenantId == surveyTenantId)
        {
            // Admin can do anything, as long as the resource belongs to the admin's tenant.
            if (context.User.HasClaim(ClaimTypes.Role, Roles.SurveyAdmin))
            {
                context.Succeed(requirement);
                return Task.FromResult(0);
            }

            if (context.User.HasClaim(ClaimTypes.Role, Roles.SurveyCreator))
            {
                permissions.Add(UserPermissionType.Creator);
            }
            else
            {
                permissions.Add(UserPermissionType.Reader);
            }

            if (resource.OwnerId == userId)
            {
                permissions.Add(UserPermissionType.Owner);
            }
        }
        if (resource.Contributors != null && resource.Contributors.Any(x => x.UserId == userId))
        {
            permissions.Add(UserPermissionType.Contributor);
        }

        if (ValidateUserPermissions[requirement](permissions))
        {
            context.Succeed(requirement);
        }
        return Task.FromResult(0);
    }
}
```

In a multitenant application, you must ensure that permissions don't "leak" to another tenant's data. In the Surveys app, the Contributor permission is allowed across tenants—you can assign someone from another tenant as a contributor. The other permission types are restricted to resources that belong to that user's tenant. To enforce this requirement, the code checks the tenant ID before granting the permission. (The `TenantId` field as assigned when the survey is created.)

The next step is to check the operation (such as read, update, or delete) against the permissions. The Surveys app implements this step by using a lookup table of functions:

```
static readonly Dictionary<OperationAuthorizationRequirement, Func<List<UserPermissionType>, bool>>
ValidateUserPermissions
    = new Dictionary<OperationAuthorizationRequirement, Func<List<UserPermissionType>, bool>>

    {
        { Operations.Create, x => x.Contains(UserPermissionType.Creator) },

        { Operations.Read, x => x.Contains(UserPermissionType.Creator) ||
                                x.Contains(UserPermissionType.Reader) ||
                                x.Contains(UserPermissionType.Contributor) ||
                                x.Contains(UserPermissionType.Owner) },

        { Operations.Update, x => x.Contains(UserPermissionType.Contributor) ||
                                x.Contains(UserPermissionType.Owner) },

        { Operations.Delete, x => x.Contains(UserPermissionType.Owner) },

        { Operations.Publish, x => x.Contains(UserPermissionType.Owner) },

        { Operations.UnPublish, x => x.Contains(UserPermissionType.Owner) }
    };
```

Next

Secure a backend web API for multitenant applications

11/19/2021 • 6 minutes to read • Edit Online

Sample code

The Tailspin Surveys application uses a backend web API to manage CRUD operations on surveys. For example, when a user clicks "My Surveys", the web application sends an HTTP request to the web API:

```
GET /users/{userId}/surveys
```

The web API returns a JSON object:

```
{
  "Published":[],
  "Own":[
    {"Id":1,"Title":"Survey 1"},
    {"Id":3,"Title":"Survey 3"},
    ],
  "Contribute": [{"Id":8,"Title":"My survey"}]
}
```

The web API does not allow anonymous requests, so the web app must authenticate itself using OAuth 2 bearer tokens.

> **NOTE**
>
> This is a server-to-server scenario. The application does not make any AJAX calls to the API from the browser client.

There are two main approaches you can take:

- Delegated user identity. The web application authenticates with the user's identity.
- Application identity. The web application authenticates with its client ID, using OAuth 2 client credential flow.

The Tailspin application implements delegated user identity. Here are the main differences:

Delegated user identity:

- The bearer token sent to the web API contains the user identity.
- The web API makes authorization decisions based on the user identity.
- The web application needs to handle 403 (Forbidden) errors from the web API, if the user is not authorized to perform an action.
- Typically, the web application still makes some authorization decisions that affect UI, such as showing or hiding UI elements).
- The web API can potentially be used by untrusted clients, such as a JavaScript application or a native client application.

Application identity:

- The web API does not get information about the user.

- The web API cannot perform any authorization based on the user identity. All authorization decisions are made by the web application.
- The web API cannot be used by an untrusted client (JavaScript or native client application).
- This approach may be somewhat simpler to implement, because there is no authorization logic in the Web API.

In either approach, the web application must get an access token, which is the credential needed to call the web API.

- For delegated user identity, the token has to come from an identity provider (IDP), such as Azure Active Directory, which can issue a token on behalf of the user.
- For client credentials, an application might get the token from the IDP or host its own token server. (But don't write a token server from scratch; use a well-tested framework like IdentityServer4.) If you authenticate with Azure AD, it's strongly recommended to get the access token from Azure AD, even with client credential flow.

The rest of this article assumes the application is authenticating with Azure AD.

A diagram that shows the web application requesting an access token from Azure AD and sending the token to the web API.

Register the web API in Azure AD

In order for Azure AD to issue a bearer token for the web API, you need to configure some things in Azure AD.

1. Register the web API in Azure AD.
2. Add the client ID of the web app to the web API application manifest, in the `knownClientApplications` property. See the GitHub readme for more information.
3. Give the web application permission to call the web API. In the Azure portal, you can set two types of permissions: "Application Permissions" for application identity (client credential flow), or "Delegated Permissions" for delegated user identity.

A screenshot of the Azure portal that shows the application permissions and delegated permissions.

Getting an access token

Before calling the web API, the web application gets an access token from Azure AD. In a .NET application, use the Microsoft Authentication Library for .NET (MSAL.NET). Add `.EnableTokenAcquisitionToCallDownstreamApi()` in Startup.cs of the application.

After acquiring a token, MSAL caches it. So, you'll also need to choose a token cache implementation, which is included in MSAL. This example uses distributed cache. For details, see See Token caching.

Using the access token to call the web API

Once you have the token, call a downstream web API. That process is described in Call a downstream web API with the helper class.

Authenticating in the web API

The web API has to authenticate the bearer token. In ASP.NET Core, you can use the Microsoft.AspNet.Authentication.JwtBearer package. This package provides middleware that enables the application to receive OpenID Connect bearer tokens.

Register the middleware in your web API `Startup` class.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddMicrosoftIdentityWebApi(jtwOptions =>
        {
            jtwOptions.Events = new
SurveysJwtBearerEvents(loggerFactory.CreateLogger<SurveysJwtBearerEvents>());
        },
        msIdentityOptions => {
            Configuration.GetSection("AzureAd").Bind(msIdentityOptions);
        });
```

Events is a class that derives from **JwtBearerEvents**.

Issuer validation

Validate the token issuer in the **JwtBearerEvents.TokenValidated** event. The issuer is sent in the "iss" claim.

In the Surveys application, the web API doesn't handle tenant sign-up. Therefore, it just checks if the issuer is already in the application database. If not, it throws an exception, which causes authentication to fail.

```
public override async Task TokenValidated(TokenValidatedContext context)
{
    var principal = context.Ticket.Principal;
    var tenantManager = context.HttpContext.RequestServices.GetService<TenantManager>();
    var userManager = context.HttpContext.RequestServices.GetService<UserManager>();
    var issuerValue = principal.GetIssuerValue();
    var tenant = await tenantManager.FindByIssuerValueAsync(issuerValue);

    if (tenant == null)
    {
        // The caller was not from a trusted issuer. Throw to block the authentication flow.
        throw new SecurityTokenValidationException();
    }

    var identity = principal.Identities.First();

    // Add new claim for survey_userid
    var registeredUser = await userManager.FindByObjectIdentifier(principal.GetObjectIdentifierValue());
    identity.AddClaim(new Claim(SurveyClaimTypes.SurveyUserIdClaimType, registeredUser.Id.ToString()));
    identity.AddClaim(new Claim(SurveyClaimTypes.SurveyTenantIdClaimType,
registeredUser.TenantId.ToString()));

    // Add new claim for Email
    var email = principal.FindFirst(ClaimTypes.Upn)?.Value;
    if (!string.IsNullOrWhiteSpace(email))
    {
        identity.AddClaim(new Claim(ClaimTypes.Email, email));
    }
}
```

As this example shows, you can also use the **TokenValidated** event to modify the claims. Remember that the claims come directly from Azure AD. If the web application modifies the claims that it gets, those changes won't show up in the bearer token that the web API receives. For more information, see Claims transformations.

Authorization

For a general discussion of authorization, see Role-based and resource-based authorization.

The JwtBearer middleware handles the authorization responses. For example, to restrict a controller action to authenticated users, use the **[Authorize]** attribute and specify **JwtBearerDefaults.AuthenticationScheme** as the authentication scheme:

```
[Authorize(ActiveAuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
```

This returns a 401 status code if the user is not authenticated.

To restrict a controller action by authorization policy, specify the policy name in the **[Authorize]** attribute:

```
[Authorize(Policy = PolicyNames.RequireSurveyCreator)]
```

This returns a 401 status code if the user is not authenticated, and 403 if the user is authenticated but not authorized. Register the policy on startup:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(PolicyNames.RequireSurveyCreator,
            policy =>
            {
                policy.AddRequirements(new SurveyCreatorRequirement());
                policy.RequireAuthenticatedUser(); // Adds DenyAnonymousAuthorizationRequirement
                policy.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme);
            });
        options.AddPolicy(PolicyNames.RequireSurveyAdmin,
            policy =>
            {
                policy.AddRequirements(new SurveyAdminRequirement());
                policy.RequireAuthenticatedUser(); // Adds DenyAnonymousAuthorizationRequirement
                policy.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme);
            });
    });

    // ...
}
```

Protecting application secrets

It's common to have application settings that are sensitive and must be protected, such as:

- Database connection strings
- Passwords
- Cryptographic keys

As a security best practice, you should never store these secrets in source control. It's too easy for them to leak — even if your source code repository is private. And it's not just about keeping secrets from the general public. On larger projects, you might want to restrict which developers and operators can access the production secrets. (Settings for test or development environments are different.)

A more secure option is to store these secrets in Azure Key Vault. Key Vault is a cloud-hosted service for managing cryptographic keys and other secrets. This article shows how to use Key Vault to store configuration settings for your app.

In the Tailspin Surveys application, the following settings are secret:

- The database connection string.
- The Redis connection string.
- The client secret for the web application.

The Surveys application loads configuration settings from the following places:

- The appsettings.json file
- The user secrets store (development environment only; for testing)
- The hosting environment (app settings in Azure web apps)
- Key Vault (when enabled)

Each of these overrides the previous one, so any settings stored in Key Vault take precedence.

> **NOTE**
> By default, the Key Vault configuration provider is disabled. It's not needed for running the application locally. You would enable it in a production deployment.

At startup, the application reads settings from every registered configuration provider, and uses them to populate a strongly typed options object. For more information, see Using Options and configuration objects.

Next

Cache access tokens

11/19/2021 • 2 minutes to read • Edit Online

Sample code

It's relatively expensive to get an OAuth access token, because it requires an HTTP request to the token endpoint. Therefore, it's good to cache tokens whenever possible. The Microsoft Authentication Library for .NET (MSAL.NET) (MSAL) caches tokens obtained from Azure AD, including refresh tokens.

Some implementations include MSAL are in-memory cache and distributed cache. This option is set in the ConfigureServices method of the Startup class of the web application. To acquire a token for the downstream API, you'll need to `.EnableTokenAcquisitionToCallDownstreamApi()`.

The Surveys app uses distributed token cache that stores data in the backing store. The app uses a Redis cache as the backing store. Every server instance in a server farm reads/writes to the same cache, and this approach scales to many users.

For a single-instance web server, you could use the ASP.NET Core in-memory cache. (This is also a good option for running the app locally during development.)

```
services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
    .AddMicrosoftIdentityWebApp(
    options =>
    {
        Configuration.Bind("AzureAd", options);
        options.Events = new SurveyAuthenticationEvents(loggerFactory);
        options.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        options.Events.OnTokenValidated += options.Events.TokenValidated;
    })
    .EnableTokenAcquisitionToCallDownstreamApi()
    .AddDownstreamWebApi(configOptions.SurveyApi.Name, Configuration.GetSection("SurveyApi"))
    .AddDistributedTokenCaches();

    services.AddStackExchangeRedisCache(options =>
    {
        options.Configuration = configOptions.Redis.Configuration;
        options.InstanceName = "TokenCache";
    });
}
```

The configuration for SurveyApi is specified in appsettings.json.

```
  "SurveyApi": {
    "BaseUrl": "https://localhost:44301",
    "Scopes": "https://test.onmicrosoft.com/surveys.webapi/surveys.access",
    "Name": "SurveyApi"
  },
```

Encrypting cached tokens

Tokens are sensitive data, because they grant access to a user's resources. (Moreover, unlike a user's password, you can't just store a hash of the token.) Therefore, it's critical to protect tokens from being compromised.

The Redis-backed cache is protected by a password, but if someone obtains the password, they could get all of

the cached access tokens. The MSAL token cache is encrypted.

Acquire the token

The Survey application calls the downstream web API from the page constructor.

```
public class SurveyService : ISurveyService
{
    private string _serviceName;
    private readonly IDownstreamWebApi _downstreamWebApi;

    public SurveyService(HttpClientService factory, IDownstreamWebApi downstreamWebApi,
IOptions<ConfigurationOptions> configOptions)
    {
        _serviceName = configOptions.Value.SurveyApi.Name;
        _downstreamWebApi = downstreamWebApi;
    }

    public async Task<SurveyDTO> GetSurveyAsync(int id)
    {
        return await _downstreamWebApi.CallWebApiForUserAsync<SurveyDTO>(_serviceName,
            options =>
            {
                options.HttpMethod = HttpMethod.Get;
                options.RelativePath = $"surveys/{id}";
            });
    }
}
```

Another way is to inject an `ITokenAcquisition` service in the controller. For more information, see Acquire and cache tokens using the Microsoft Authentication Library (MSAL)

Next

Next steps

- Token cache serialization in MSAL.NET
- Acquire and cache tokens using the Microsoft Authentication Library (MSAL)

Related resources

- Identity management in multitenant applications
- Secure a backend web API for multitenant applications

Use client certificate to get access tokens from Azure AD

11/19/2021 • 2 minutes to read • Edit Online

Sample code

This article describes how to add client certificate to the Tailspin Surveys sample application.

When using authorization code flow or hybrid flow in OpenID Connect, the client exchanges an authorization code for an access token. During this step, the client has to authenticate itself to the server.

Example: Hybrid flow

There are many ways to authenticate the client, using client secret, certificate, and assertions. The Tailspin Surveys application is configured to use client secret by default.

Here is an example request from the client to the IDP, requesting an access token. Note the `client_secret` parameter.

```
POST https://login.microsoftonline.com/b9bd2162xxx/oauth2/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

resource=https://tailspin.onmicrosoft.com/surveys.webapi
  &client_id=87df91dc-63de-4765-8701-b59cc8bd9e11
  &client_secret=i3Bf12Dn...
  &grant_type=authorization_code
  &code=PG8wJG6Y...
```

The secret is just a string, so you have to make sure not to leak the value. The best practice is to keep the client secret out of source control. When you deploy to Azure, store the secret in an app setting.

However, anyone with access to the Azure subscription can view the app settings. Furthermore, there is always a temptation to check secrets into source control (for example, in deployment scripts), share them by email, and so on.

For additional security, you can use a client certificate instead of a client secret. The client uses a certificate to prove the token request came from the client. The client certificate is stored in key vault. For this option, add the `ClientCertificates` under AzureAd and specify the configuration settings as shown here:

```
  "ClientCertificates": [
    {
      "SourceType": "KeyVault",
      "KeyVaultUrl": "https://msidentitywebsamples.vault.azure.net",
      "KeyVaultCertificateName": "MicrosoftIdentityCert"
    }
   ]
```

NOTE

For more information, see Using certificates with Microsoft.Identity.Web.

Next

Federate with a customer's AD FS

11/19/2021 • 6 minutes to read • Edit Online

This article describes how a multitenant SaaS application can support authentication via Active Directory Federation Services (AD FS), in order to federate with a customer's AD FS.

Federation scenario

Azure Active Directory (Azure AD) makes it easy to sign in users from Azure AD tenants, including Office365 and Dynamics CRM Online customers. But what about customers who use on-premises Active Directory on a corporate intranet?

One option is for these customers to sync their on-premises AD with Azure AD, using Azure AD Connect. However, some customers may be unable to use this approach, due to corporate IT policy or other reasons. In that case, another option is to federate through Active Directory Federation Services (AD FS).

To enable this scenario:

- The customer must have an Internet-facing AD FS farm.
- The SaaS provider deploys their own AD FS farm.
- The customer and the SaaS provider must set up federation trust. This is a manual process.

There are three main roles in the trust relation:

- The customer's AD FS is the account partner, responsible for authenticating users from the customer's AD, and creating security tokens with user claims.
- The SaaS provider's AD FS is the resource partner, which trusts the account partner and receives the user claims.
- The application is configured as a relying party (RP) in the SaaS provider's AD FS.

> **NOTE**
>
> In this article, we assume the application uses OpenID Connect as the authentication protocol. Another option is to use WS-Federation.
>
> For OpenID Connect, the SaaS provider must use AD FS 2016, running in Windows Server 2016. AD FS 3.0 does not support OpenID Connect.

For an example of using WS-Federation with ASP.NET 4, see the active-directory-dotnet-webapp-wsfederation sample.

Authentication flow

1. When the user clicks "sign in", the application redirects to an OpenID Connect endpoint on the SaaS provider's AD FS.
2. The user enters his or her organizational user name ("`alice@corp.contoso.com`"). AD FS uses home realm discovery to redirect to the customer's AD FS, where the user enters their credentials.
3. The customer's AD FS sends user claims to the SaaS provider's AD FS, using WF-Federation (or SAML).
4. Claims flow from AD FS to the app, using OpenID Connect. This requires a protocol transition from WS-Federation.

Limitations

By default, the relying party application receives only a fixed set of claims available in the id_token, shown in the following table. With AD FS 2016, you can customize the id_token in OpenID Connect scenarios. For more information, see Custom ID Tokens in AD FS.

CLAIM	DESCRIPTION
aud	Audience. The application for which the claims were issued.
authenticationinstant	Authentication instant. The time at which authentication occurred.
c_hash	Code hash value. This is a hash of the token contents.
exp	Expiration time. The time after which the token will no longer be accepted.
iat	Issued at. The time when the token was issued.
iss	Issuer. The value of this claim is always the resource partner's AD FS.
name	User name. Example: `john@corp.fabrikam.com`
nameidentifier	Name identifier. The identifier for the name of the entity for which the token was issued.
nonce	Session nonce. A unique value generated by AD FS to help prevent replay attacks.
upn	User principal name (UPN). Example: `john@corp.fabrikam.com`
pwd_exp	Password expiration period. The number of seconds until the user's password or a similar authentication secret, such as a PIN. expires.

> **NOTE**
> The "iss" claim contains the AD FS of the partner (typically, this claim will identify the SaaS provider as the issuer). It does not identify the customer's AD FS. You can find the customer's domain as part of the UPN.

The rest of this article describes how to set up the trust relationship between the RP (the app) and the account partner (the customer).

AD FS deployment

The SaaS provider can deploy AD FS either on-premises or on Azure VMs. For security and availability, the following guidelines are important:

- Deploy at least two AD FS servers and two AD FS proxy servers to achieve the best availability of the AD FS service.
- Domain controllers and AD FS servers should never be exposed directly to the Internet and should be in a virtual network with direct access to them.
- Web application proxies (previously AD FS proxies) must be used to publish AD FS servers to the Internet.

To set up a similar topology in Azure requires the use of virtual networks, network security groups, virtual machines, and availability sets. For more details, see Guidelines for deploying Windows Server Active Directory on Azure Virtual Machines.

Configure OpenID Connect authentication with AD FS

The SaaS provider must enable OpenID Connect between the application and AD FS. To do so, add an application group in AD FS. You can find detailed instructions in this blog post, under "Setting up a Web App for OpenId Connect sign in AD FS."

Next, configure the OpenID Connect middleware. The metadata endpoint is `https://domain/adfs/.well-known/openid-configuration`, where domain is the SaaS provider's AD FS domain.

Typically you might combine this with other OpenID Connect endpoints (such as Azure AD). You'll need two different sign-in buttons or some other way to distinguish them, so that the user is sent to the correct authentication endpoint.

Configure the AD FS Resource Partner

The SaaS provider must do the following for each customer that wants to connect via AD FS:

1. Add a claims provider trust.
2. Add claims rules.
3. Enable home-realm discovery.

Here are the steps in more detail.

Add the claims provider trust

1. In Server Manager, click **Tools**, and then select **AD FS Management**.
2. In the console tree, under **AD FS**, right-click **Claims Provider Trusts**. Select **Add Claims Provider Trust**.
3. Click **Start** to start the wizard.
4. Select the option "Import data about the claims provider published online or on a local network". Enter the URI of the customer's federation metadata endpoint. (Example: `https://contoso.com/FederationMetadata/2007-06/FederationMetadata.xml`.) You will need to get this from the customer.

5. Complete the wizard using the default options.

Edit claims rules

1. Right-click the newly added claims provider trust, and select **Edit Claims Rules**.
2. Click **Add Rule**.
3. Select "Pass Through or Filter an Incoming Claim" and click **Next**.

4. Enter a name for the rule.
5. Under "Incoming claim type", select **UPN**.
6. Select "Pass through all claim values".

7. Click **Finish**.
8. Repeat steps 2 - 7, and specify **Anchor Claim Type** for the incoming claim type.
9. Click **OK** to complete the wizard.

Enable home-realm discovery

Run the following PowerShell script:

```
Set-AdfsClaimsProviderTrust -TargetName "name" -OrganizationalAccountSuffix @("suffix")
```

where "name" is the friendly name of the claims provider trust, and "suffix" is the UPN suffix for the customer's AD (example, "corp.fabrikam.com").

With this configuration, end users can type in their organizational account, and AD FS automatically selects the corresponding claims provider. See Customizing the AD FS Sign-in Pages, under the section "Configure Identity Provider to use certain email suffixes".

Configure the AD FS Account Partner

The customer must do the following:

1. Add a relying party (RP) trust.
2. Adds claims rules.

Add the RP trust

1. In Server Manager, click **Tools**, and then select **AD FS Management**.
2. In the console tree, under **AD FS**, right-click **Relying Party Trusts**. Select **Add Relying Party Trust**.
3. Select **Claims Aware** and click **Start**.
4. On the **Select Data Source** page, select the option "Import data about the claims provider published online

or on a local network". Enter the URI of the SaaS provider's federation metadata endpoint.

5. On the **Specify Display Name** page, enter any name.
6. On the **Choose Access Control Policy** page, choose a policy. You could permit everyone in the organization, or choose a specific security group.

7. Enter any parameters required in the **Policy** box.
8. Click **Next** to complete the wizard.

Add claims rules

1. Right-click the newly added relying party trust, and select **Edit Claim Issuance Policy**.
2. Click **Add Rule**.
3. Select "Send LDAP Attributes as Claims" and click **Next**.
4. Enter a name for the rule, such as "UPN".
5. Under **Attribute store**, select **Active Directory**.

6. In the **Mapping of LDAP attributes** section:

- Under **LDAP Attribute**, select **User-Principal-Name**.
- Under **Outgoing Claim Type**, select **UPN**.

7. Click **Finish**.

8. Click **Add Rule** again.
9. Select "Send Claims Using a Custom Rule" and click **Next**.
10. Enter a name for the rule, such as "Anchor Claim Type".
11. Under **Custom rule**, enter the following:

    ```
    EXISTS([Type == "http://schemas.microsoft.com/ws/2014/01/identity/claims/anchorclaimtype"])=>
    issue (Type = "http://schemas.microsoft.com/ws/2014/01/identity/claims/anchorclaimtype",
          Value = "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/upn");
    ```

    This rule issues a claim of type `anchorclaimtype`. The claim tells the relying party to use UPN as the user's immutable ID.
12. Click **Finish**.
13. Click **OK** to complete the wizard.

Secure development with single-page applications (SPAs)

11/19/2021 • 8 minutes to read • Edit Online

When developing cloud-native distributed systems, securing such systems can introduce a new layer of complexity.

On-premise systems rely on the security boundaries that the internal network provides, and they use the directory services for user security. They can run for many years within this secure environment without problems. Moving to the cloud can present new security risks. This article describes tools that you can use to mitigate these risks.

One such tool is access control. Access control identifies users and regulates what they can do when interacting with an application.

There are two parts to access control:

- **Authentication** identifies the user.
- **Authorization** determines what the user can do in the application.

OAuth, an open framework, helps address these challenges and provides a protocol for developers to use when building their systems. OAuth 2.0 is the current standard.

OAuth 2.0 provides secure delegated access. By issuing access tokens, you can authorize third-party access to your protected resources without providing credentials.

Azure Active Directory (Azure AD) is Microsoft's built-in solution for managing identities in the cloud. It integrates with on-premise systems so that users have a seamless experience when accessing protect services in the cloud.

This guide shows you how to use Azure AD and OAuth 2.0 to secure a single-page application.

OAuth flows

OAuth flows cover many use cases, all backed by Azure AD Services. Developers use these flows to build a secure application, so that:

- Users can securely access client systems.
- Guest users can participate through business-to-business transactions.
- Users can reach out to end consumers through Azure Business to Consumers (Azure B2C).

There are two OAuth flows, implicit grant and authorization code. Implicit grant is the most common, but we recommend using the authorization code flow.

Register your application in Azure

Register a service principal for the UI and API using Azure AD Directory in the Azure portal.

1. Log in to the Azure portal, and then search for **App registrations**.
2. Select **New registration**.

3. To register a new application, you need:
   - The display name for the application.
   - The supported account type.
   - The application type: Web, SPA, or public client/native (mobile and desktop).
   - The redirect URI. When the user is authenticated, Azure AD redirects the result to the client.
     - An example for local development is http://localhost:4200.
     - An example for production is 'https://portal.contoso.com'.

4. Select **Register**.

5. Once the registration is complete, select **Overview**, and then select your application name next to **Managed application in local directory**.

6. Select **Properties**, switch **User assignment required** to **Yes** to set the access permissions for the application, and then select **Save**.

7. Select **Users and groups**, and then add existing or new users and security groups.

8. Your users can access the application through My Apps.

Set up configuration details in the client application

After you create and configure the app registration in Azure, you can set up the configuration details in the client application. For a single-page framework like Angular, Microsoft has developed the **@Azure/msal-angular** library to help you integrate Azure AD in your client application.

1. Install an @Azure/msal-angular library.

2. Configure the library.

   - The `protectedResourceMap` contains a list of protected resources and their scopes in an array: [[protected resource], [scopes for resource]].
   - The `clientID` and `authority`, which is the tenant ID, are supplied to the configuration object.
   - For protected HTTP requests, the client inserts a new header property called Authorization. It contains the bearer token for the authenticated user. The bearer token gives the downstream OAuth 2.0 service a secure point of entry. It can include metadata for the service when authorizing the request.

```
export const protectedResourceMap: [string, string[]][]] = [
    ['https://graph.microsoft.com/v1.0/me', ['user.read']],
    ['https://localhost:5001/api/weatherforecast', ['api://ae05da8f-07d0-4ae6-aef1-
18a6af68e5dd/access_as_user']]
];

function MSALConfigFactory(): Configuration {
    return {
        auth: {
            clientId: 'eba23c0b-1e86-4f68-b1d2-9c54d96083de',
            authority: 'https://login.microsoftonline.com/1c302616-bc6a-45a6-9c07-838c89d55003',
            redirectUri: 'http://localhost:4200',
            validateAuthority: true,
            postLogoutRedirectUri: 'http://localhost:4200',
            navigateToLoginRequestUrl: true
        },
        cache: {
            cacheLocation: 'sessionStorage',

            storeAuthStateInCookie: false //set to false, not ie 11
        }
    };
}
```

For more information on configuring an Angular library, see Tutorial: Sign in users and call the Microsoft Graph API from an Angular single-page application (SPA) using auth code flow.

Test the application authentication

Test the authentication process by having a user *with* access, and a user *without* access attempt to log in to the client.

The user logs in to the application and is redirected to their Azure AD tenant.

- If the user is valid, they're authenticated and logged in.
- If the user isn't valid, the application returns an error.

Consume a protected resource or resource server

To consume a protected resource, create another app registration. After the app registration is complete, the API changes the bearer token to allow access.

Expose the API

1. Create another app registration in Azure.
2. Select **Expose an API**, and then select **Add a scope**.

3. Enter the **Application ID URI**, and then select **Save and continue**. This permission is used by the API to validate the request.

4. Configure the scope name and consent information. If you select **Admins only**, only Admins can grant consent for the directory.

Add the API to the App registration

Now that you've defined your permissions and exposed the API, you need to add the API to the app registration for the client.

1. In your app registration, select **API permissions**, and then **Add a permission**.

2. Select **My APIs**, and then select the API registration you created.

3. Select the scope you created to expose the API permission, and then select **Add permissions**.

Now the API is added to the application. Since you might need to grant consent again for access to the API, consider granting admin consent so that users don't have to reconsent.

Add the API to the protected resource map

Now that the configuration in the Azure portal is complete, the UI client can consume the resource. Add the API to the protected resource map to make sure that the UI attaches the correct bearer token for the API request.

```
export const protectedResourceMap: [string, string[]][] = [
    ['https://graph.microsoft.com/v1.0/me', ['user.read']],
    ['https://localhost:5001/api/weatherforecast', ['api://eba23c0b-1e86-4f68-b1d2-
9c54d96083de/access_as_user']]
];
```

When your client application attempts to access the resource, the MSAL Client Library authenticates to Azure AD through a hidden iframe, and then returns a bearer token for the resource. The bearer token is only added for requests that match the endpoint, in this case https://localhost:5001/api/weatherforecast.

If the API you configured with the relevant app registrations receives a bearer token with an invalid application

ID URI, it rejects the request and returns a 401 unauthorized message.

In the following example, the backend service is written in .NET Core. The example shows the configuration properties for the API. The `ClientId` has the application ID URI in the form of `api://{clientId}`.

```
"AzureAD": {
  "Instance": "https://login.microsoftonline.com/",
  "Domain": "yourName.onmicrosoft.com",
  "TenantId": "1c302616-bc6a-45a6-9c07-838c89d55003",
  "ClientId": "api://ae05da8f-07d0-4ae6-aef1-18a6af68e5dd"
},
```

Within the startup class of the .NET Core API, the Authentication scheme and options are added to the configure services method.

```
services.Addauthentication(AzureADDefaults.BearerAuthenticationScheme).AddAzureADBearer(options =>
Configuration.Bind("AzureAD",
  options));
```

When the client calls the API, the bearer token gets added to the request.

You can navigate to jwt.ms and paste the bearer token into a human-readable format.

You can see that the API URI is inside the `aud` property. This property identifies the intended recipient of the token, which is your API. If your API is not the intended recipient, it automatically rejects the request with a 401 HTTP response.

The `scp` property contains the set of scopes exposed by your application. If invalid scopes are added through the client, Azure AD returns an error requesting further authorization for scope.

Decoded Token | Claims


```
{
  "typ": "JWT",
  "alg": "RS256",
  "x5t": "SsZsBNhZcF3Q9S4trpQBTByNRRI",
  "kid": "SsZsBNhZcF3Q9S4trpQBTByNRRI"
}.{
  "aud": "api://ae05da8f-07d0-4ae6-aef1-18a6af68e5dd",
  "iss": "https://sts.windows.net/1c302616-bc6a-45a6-9c07-838c89d55003/",
  "iat": 1593442033,
  "nbf": 1593442033,
  "exp": 1593445933,
  "acr": "1",
  "aio": "ATQAy/8QAAAAjEApcCONsUrKaL8eogaACk63EyCCpBXXqRPqzVTAollISoQzinZdUVkFor+svPUR",
  "amr": [
    "pwd"
  ],
  "appid": "eba23c0b-1e86-4f68-b1d2-9c54d96083de",
  "appidacr": "0",
  "groups": [
    "5cbf7ec8-546c-410a-80e9-e39723d2fa28"
  ],
  "ipaddr": "31.125.2.173",
  "name": "lavanuser",
  "oid": "fd6b5216-925a-40ea-b353-947c586927fa",
  "scp": "access_as_user",
  "sub": "8mu-2rBDb4eNxsmAqmjlDtqR5DBx9KRF1p3ozq_wHQc",
  "tid": "1c302616-bc6a-45a6-9c07-838c89d55003",
  "unique_name": "lavanuser@lavanngmail.onmicrosoft.com",
  "upn": "lavanuser@lavanngmail.onmicrosoft.com",
  "uti": "kWcsJcnLa0-rVrE5E0-nAQ",
  "ver": "1.0"
}.[Signature]
```


Use the application manifest to further define authorization

Further authorization practices are implemented by using the application manifest for the API app registration. Since you have explicitly defined users, you can add further levels of authorization and only allow members of a specific security group to access more sensitive resources.

1. In your app registration, select **Manifest**.

2. Edit the key value pairs of the JSON object as needed.

In general, it's best to only issue `SecurityGroup`. If you use `All`, security groups, Azure AD roles, and distribution groups are emitted. The token has a set limit of 200, and if that limit is reached, an overage claim is added. The claim points to the Graph endpoint to retrieve the list of groups for the user.

After configuration, the jwt token has a new property, `groups`, that contains the unique object IDs that can be used to enforce authorization.

The API can be configured with a policy that looks for a required claim and value for role-based authorization through the policy handler.

```
public void ConfigureServices(IServiceCollection services)
{

    services.AddAuthentication(AzureADDefaults.BearerAuthenticationScheme).AddAzureADBearer(options =>
Configuration.Bind("AzureAD",
      options));

    services.AddAuthorization(options =>
    {

        options.AddPolicy("DensuAegisReportsAdmin", policyBuilder =>
        {
            policyBuilder.RequireClaim("groups", "ebde25e7-d254-474e-ae33-cd491aa98ebf"); //This would be an
environment variable
        });
});

    JWTSecurityTokenHandler.DefaultMapInboundClaims = false;
    services.AddCors();

    services.AddControllers();
}
```

The controller for the API can have the relevant attributions added. These attributes offer more security and help confirm the authenticated persons are authorized to access the protected resource.

```
[Route("admin")]
[Authorize("DensuAegisReportsAdmin")]
public IActionResult GetForcastsForAdmin()
{
    var user = User.Claims;

    var groups = User.Claims.Where(c => c.Type == "groups").Select(c => c.Value).ToList();
    var userName = UserClaims.Where(c => c.Type == "unique_name").Select(c => c.Value).FirstOrDefault();
    // SecurityGroup = groups
    var rng = new Random();
    var forecasts = Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)],

    })
    .ToArray();

    return Ok(new
    {
        User = userName
        ,
        SecurityGroup = groups
        ,
        Forcasts = forecasts
    });
}
```

More roles can be created with the application manifest that are unique to the app registration. Then, more groups can be created within the context of the application.

For example, you can create a custom role called AppAdmin that is unique to the application registration. Using the enterprise application build, you can assign users or security groups to that role.

When you call the protected resource after the configuration change, the bearer token has the `roles` property inside the bearer token.


```
Decoded Token   Claims

{
  "typ": "JWT",
  "alg": "RS256",
  "x5t": "SsZsBNhZcF3Q9S4trpQBTByNRRI",
  "kid": "SsZsBNhZcF3Q9S4trpQBTByNRRI"
}.{
  "aud": "api://6427c716-86eb-4750-ba32-f7b9a9ef3e55",
  "iss": "https://sts.windows.net/1c302616-bc6a-45a6-9c07-838c89d55003/",
  "iat": 1593444164,
  "nbf": 1593444164,
  "exp": 1593448064,
  "acr": "1",
  "aio": "ATQAy/8QAAAA7AGRZ19Cro34PsnmaKsQvVzcx6kCuZ/+h6pP7ZTShEFUVTHwWy7azEeTnt7a/QII",
  "amr": [
    "pwd"
  ],
  "appid": "6ec56496-5897-4cc4-bfe9-c593c19feda3",
  "appidacr": "0",
  "groups": [
    "8564e9ef-b8b4-4661-af47-f6c728516e97",
    "5cbf7ec8-546c-410a-80e9-e39723d2fa28"
  ],
  "ipaddr": "31.125.2.173",
  "name": "lavann",
  "oid": "5cceed7b-5f35-4b41-ae6b-07ed8a47aa2c",
  "roles": [
    "AppAdmin"
  ],
  "scp": "access_as_user",
  "sub": "P6S-nVGZ_ysiBqhGy50YImWGtFFUb4WqHJXXy5AYAsU",
  "tid": "1c302616-bc6a-45a6-9c07-838c89d55003",
  "unique_name": "lavan@lavanngmail.onmicrosoft.com",
  "upn": "lavan@lavanngmail.onmicrosoft.com",
  "uti": "TTSB1HeB7ESNfTMEThjBAQ",
  "ver": "1.0"
}.[Signature]
```


The API is configured using the policy builder under **Configure Services**.

```
// Adding authorization policies that enforce authorization using Azure AD roles.
services.AddAuthorization(options =>
{
    options.AddPolicy(AuthorizationPolicies.AssignmentToAppAdminRoleRequired, policy =>
      policy.RequireRole(AppRole.AppAdmin));
});
```

The protected route uses the authorization policy to make sure that the authenticated user is in the relevant role

before authorizing the request.

Next steps

- Integrate on-premises AD domains with Azure AD
- Azure Active Directory identity management and access management for AWS
- Deploy AD DS in an Azure virtual network

Related resources

- OAuth 2.0 and OpenID Connect protocols on the Microsoft identity platform
- Microsoft identity platform and implicit grant flow
- Microsoft identity platform and OAuth 2.0 authorization code flow

Getting started with Azure IoT solutions

11/19/2021 • 3 minutes to read • Edit Online

IoT (Internet of Things) is a collection of managed and platform services that connect and control IoT assets. For example, consider an industrial motor connected to the cloud. The motor collects and sends temperature data. This data is used to evaluate whether the motor is performing as expected. This information can then be used to prioritize a maintenance schedule for the motor.

Azure IoT supports a large range of devices, including industrial equipment, microcontrollers, sensors, and so on. When connected to the cloud, these devices can send data to your IoT solution. The data can then be processed to gain insights about the device. You can use these insights to monitor, manage, and control your environment.

IoT solution concepts

Learn about Azure IoT

If you are interested in learning about Azure IoT concepts in detail using a sandbox subscription, see the Introduction to Azure IoT. This is a five hour learning path that consists of eight training modules.

Introduction to Azure IoT

Understanding the difference between PaaS and aPaaS solutions

Microsoft enables you to create an IoT solution, by using individual PaaS services or in an aPaaS IoT solution platform. PaaS (platform as a service) is a cloud computing model in which Microsoft delivers Azure hardware and software tools that are tailored to a specific task or job function. With PaaS services, you are responsible for scaling and configuration, but the underlying infrastructure as a service (IaaS) is taken care of for you. aPaaS (application platform as a service) provides a cloud environment to build, manage, and deliver applications to customers. aPaaS offerings take care of scaling and most of the configuration, but they still require developer input to build out a finished solution.

Start with Azure IoT Central (aPaaS)

Microsoft's aPaaS IoT solution platform is **Azure IoT Central**. We recommend IoT Central as a starting point for all customers. It is designed to simplify and accelerate IoT solution assembly and operations, by preassembling PaaS services from the IoT Platform, and across Azure, which are needed to build enterprise grade IoT solutions. The result is an out-of-the-box and ready-to-use UX and API surface area that is complete with the capabilities needed to connect, manage, and operate fleets of devices at scale.

Azure IoT Central

Microsoft also enables you to build solutions using a collection of Azure PaaS components. You can use these to connect and provision things, analyze the insights they collect, and act influenced by those insights.

Compare IoT Central (aPaaS) to an Azure PaaS solution approach based on your solution needs.

Compare solution approaches

Design an IoT architecture

A standard IoT solution architecture consists of five basic elements:

- **Devices** consist of industrial equipment, sensors, and microcontrollers that connect with the cloud to send and receive data.
- **Provisioning** connects the devices to the cloud.
- **Processing** analyzes data from devices to gather insights.
- **Business integration** performs actions based on insights from the device data.
- **Security monitoring** provides an end-to-end security solution for IoT workloads. Use Microsoft Defender for IoT.

These fundamentals exist whether you are using a PaaS or aPaaS solution. However, we recommend all customers start with the aPaaS solution offered by Microsoft, Azure IoT Central. Its out-of-the-box UX and API surface simplify device connectivity, operations, and management so that you can spend more time and budget using IoT data to create business value.

Azure IoT reference architecture

Azure IoT Central architecture

Monitor and optimize your Azure IoT solution

You can use the insights gathered from your device data to monitor, manage, and control your environment. You can use services such as **Power BI** to visualize and inspect your data or services such as **Azure Logic Apps** and **Microsoft Flow** to set up automated actions.

Next steps

- Azure IoT documentation
- Azure IoT Central documentation
- Azure IoT Hub
- Azure IoT Hub Device Provisioning Service
- Azure IoT Edge documentation

Related resources

See the related IoT architecture guides:

- IoT solutions conceptual overview
- Choose an Internet of Things (IoT) solution in Azure
- Vision with Azure IoT Edge
- Azure Industrial IoT Analytics Guidance

See the related IoT reference architectures and example scenarios:

- Azure IoT reference architecture
- End-to-end manufacturing using computer vision on the edge
- IoT and data analytics
- IoT using Cosmos DB
- Retail - Buy online, pickup in store (BOPIS)
- Predictive maintenance with the intelligent IoT Edge

See the related IoT solution ideas:

- Condition Monitoring for Industrial IoT
- Contactless IoT interfaces with Azure intelligent edge
- COVID-19 safe environments with IoT Edge monitoring and alerting

- Environment monitoring and supply chain optimization with IoT
- IoT connected light, power, and internet for emerging markets
- UVEN smart and secure disinfection and lighting
- Mining equipment monitoring
- Predictive Maintenance for Industrial IoT
- Process real-time vehicle data using IoT
- Cognizant Safe Buildings with IoT and Azure

Vision with Azure IoT Edge

11/19/2021 • 4 minutes to read • Edit Online

Visual inspection of products, resources, and environments has been a core practice for most enterprises, and was until recently, a very manual process. An individual, or a group of individuals, would be responsible for manually inspecting the assets or the environment. Depending on the circumstances, this could become inefficient, inaccurate, or both, due to human error and limitations.

To improve the efficacy of visual inspection, enterprises began turning to deep learning artificial neural networks known as *convolutional neural networks* (or CNNs), to emulate human vision for analysis of images and video. Today this is commonly called computer vision, or simply *Vision AI*. Artificial intelligence for image analytics spans a wide variety of industries, including manufacturing, retail, healthcare, and the public sector, and an equally wide area of use cases.

- **Vision for quality assurance:** In manufacturing environments, Vision AI can be very helpful with quality inspection of parts and processes with a high degree of accuracy and velocity. An enterprise pursuing this path automates the inspection of a product for defects to answer questions such as:
  - Is the manufacturing process producing consistent results?
  - Is the product assembled properly?
  - Can there be an earlier notification of a defect to reduce waste?
  - How to leverage drift in the computer vision model to prescribe predictive maintenance?
- **Vision for safety:** In any environment, safety is a fundamental concern for every enterprise, and the reduction of risk is a driving force for adopting Vision AI. Automated monitoring of video feeds to scan for potential safety issues provides critical time to respond to incidents, and opportunities to reduce exposure to risk. Enterprises looking at Vision AI for this use case are commonly trying to answer questions such as:
  - How compliant is the workforce with using personal protective equipment?
  - How often are people entering unauthorized work zones?
  - Are products being stored in a safe manner?
  - Are there unreported close calls in a facility, or pedestrian/equipment "near misses"?

Why vision on the Edge

Over the past decade, computer vision has become a rapidly evolving area of focus for enterprises, as cloud-native technologies such as containerization, have enabled portability and migration of this technology towards the network edge. For instance, custom vision inference models trained in the cloud can be easily containerized for use in an Azure IoT Edge runtime-enabled device.

The rationale behind migrating workloads from the cloud to the edge for Vision AI generally falls into two categories – performance and cost.

On the performance side of the equation, exfiltrating large quantities of data can cause an unintended performance strain on existing network infrastructure. Additionally, the latency of sending images and/or video streams to the cloud to retrieve results may not meet the needs of the use case. For instance, a person straying into an unauthorized area may require immediate intervention, and that scenario can affect latency when every second counts. Positioning the inferencing model near the point of ingestion, allows for near real-time scoring of the image. It also allows alerting to be performed either locally or through the cloud, depending on the network topology.

In terms of cost, sending all of the data to the cloud for analysis could significantly impact the ROI or return on investment of a Vision AI initiative. With Azure IoT Edge, a Vision AI module can be designed to only capture the relevant images with a reasonable confidence level based on the scoring. This significantly limits the amount of data being sent.

Camera considerations

Camera is understandably a very important component of an Azure IoT Edge Vision solution. To learn what considerations should be taken for this component, proceed to Camera selection in Azure IoT Edge Vision.

Hardware acceleration

To bring AI to the edge, the edge hardware should be able to run the powerful AI algorithms. To know the hardware capabilities required for IoT Edge Vision, proceed to Hardware acceleration in Azure IoT Edge Vision.

Machine learning

Machine learning can be challenging for the data on the edge, due to resource restrictions of edge devices, limited energy budget, and low compute capabilities. See Machine learning and data science in Azure IoT Edge Vision to understand the key considerations in designing the machine learning capabilities of your IoT Edge Vision solution.

Image storage

Your IoT Edge Vision solution cannot be complete without careful consideration of how and where the images generated will be stored. Read Image storage and management in Azure IoT Edge Vision for a thorough discussion.

Alerts

Your IoT Edge device may need to respond to various alerts in its environment. See Alert persistence in Azure IoT Edge Vision to understand the best practices in managing these alerts.

User interface

The user interface or UI of your IoT Edge Vision solution will vary based on the target user. The article User interface in Azure IoT Edge Vision discusses the main UI considerations.

Next steps

This series of articles demonstrate how to build a complete vision workload using Azure IoT Edge devices. For further information, you may refer to the product documentation as following:

- Azure IoT Edge documentation
- Tutorial: Perform image classification at the edge with Custom Vision Service
- Azure Machine Learning documentation
- Azure Kinect DK documentation
- MMdnn tool
- ONNX

Camera selection in Azure IoT Edge Vision

11/19/2021 • 8 minutes to read • Edit Online

One of the most critical components in any AI Vision workload is selecting the right camera. The items being identified by this camera must be presented in such a way that the artificial intelligence or machine learning models can evaluate them correctly. An in-depth understanding of the different camera types is required to understand this concept.

> **NOTE**
>
> There are different manufacturers for **area**, **line**, and **smart** cameras. Instead of recommending any one vendor over another, Microsoft recommends that you select a vendor that fits your specific needs.

Types of cameras

Area scan cameras

This camera type generates the traditional camera image, where a 2D image is captured and then sent over to the Edge hardware to be evaluated. This camera typically has a matrix of pixel sensors.

As the name suggests, area scan cameras look at a large area and are great at detecting change in an area. Examples of workloads that could use an area scan camera would be workplace safety, or detecting or counting objects (people, animals, cars, and so on) in an environment.

Examples of manufacturers of area scan cameras are Basler, Axis, Sony, Bosch, FLIR, Allied Vision.

Line scan cameras

Unlike the area scan cameras, the line scan camera has a single row of linear pixel sensors. This allows the camera to take one-pixel width images in quick successions, and then stitches them together into a video stream. This video stream is then sent over to an Edge device for processing.

Line scan cameras are great for vision workloads where the items to be identified are either moving past the camera, or need to be rotated to detect defects. The line scan camera would then be able to produce a continuous image stream for evaluation. Examples of workloads that would work best with a line scan camera are:

- an item defect detection on parts that are moved on a conveyer belt,
- workloads that require spinning to see a cylindrical object, or
- any workload that requires rotation.

Examples of manufacturers of line scan cameras are Basler, Teledyne Dalsa, Hamamatsu Corporation, DataLogic, Vieworks, and Xenics.

Embedded smart cameras

This type of camera can use either an area scan or a line scan camera for capturing the images, although a line scan smart camera is rare. An embedded smart camera can not only acquire an image, but can also process that image as it is a self-contained stand-alone system. They typically have either an RS232 or an Ethernet port output, which allows them to be integrated directly into a PLC or other IIoT interfaces.

Examples of manufacturers of embedded smart cameras are Basler, Lesuze Electronics.

Camera features

Sensor size

This is one of the most important factors to evaluate in any vision workload. A sensor is the hardware within a camera that captures the light and converts into signals, which then produce an image. The sensor contains millions of semiconducting photodetectors called photosites. A higher megapixel count does not always result in a better image. For example, let's look at two different sensor sizes for a 12-megapixel camera. Camera A has a ½ inch sensor with 12 million photosites and camera B has a 1-inch sensor with 12 million photosites. In the same lighting conditions, the camera that has the 1-inch sensor will be cleaner and sharper. Many cameras typically used in vision workloads have a sensor sized between ¼ inch to 1 inch. In some cases, much larger sensors might be required.

If a camera has a choice between a larger sensor or a smaller sensor, some factors deciding why you might choose the larger sensor are:

- need for precision measurements,
- lower light conditions,
- shorter exposure times, or fast-moving items.

Resolution

This is another important factor to both line scan and area scan camera workloads. If your workload must identify fine features, such as the writing on an IC chip, then you need greater resolution cameras. If your workload is trying to detect a face, then higher resolution is required. And if you need to identify a vehicle from a distance, again a higher resolution will be required.

Speed

Sensors come in two types- *CCD* and *CMOS*. If the vision workload requires high number of images to be captured per second, then two factors will come into play. The first is how fast is the connection on the interface of the camera. The second is what type of sensor it is. CMOS sensors have a direct readout from the photosites, because of which they typically offer a higher frame rate.

> **NOTE**
> There are several other camera features to consider when selecting the correct camera for your vision workload. These include lens selection, focal length, monochrome, color depth, stereo depth, triggers, physical size, and support. Sensor manufacturers can help you understand the specific feature that your application may require.

Camera placement

The items that you are capturing in your vision workload will determine the location and angles that the camera should be placed. The camera location can also affect the sensor type, lens type, and camera body type.

There are several different factors that can weigh into the overall decision for camera placement. Two of the most critical ones are the lighting and the field of view.

Camera lighting

In a computer vision workload, lighting is a critical component to camera placement. There are several different lighting conditions. While some of the lighting conditions would be useful for one vision workload, it might produce an undesirable effect in another. Types of lighting that are commonly used in computer vision workloads are:

- **Direct lighting:** This is the most commonly used lighting condition. This light source is projected at the object to be captured for evaluation.

- **Line lighting:** This is a single array of lights that are most used with line scan camera applications. This creates a single line of light at the focus of the camera.
- **Diffused lighting:** This type of lighting is used to illuminate an object but prevent harsh shadows and is mostly used around specular objects.
- **Back lighting:** This type of light source is used behind the object, producing a silhouette of the object. This is most useful when taking measurements, edge detection, or object orientation.
- **Axial diffused lighting:** This type of light source is often used with highly reflective objects, or to prevent shadows on the part that will be captured for evaluation.
- **Custom Grid lighting:** This is a structured lighting condition that lays out a grid of light on the object, the intent is to have a known grid projection to then provide more accurate measurements of components, parts, placement of items, and so on.
- **Strobe lighting:** Strobe lighting is used for high speed moving parts. The strobe must be in sync with the camera to take a *freeze* of the object for evaluation, this lighting helps to prevent motion blurring effect.
- **Dark Field lighting:** This type of light source uses several lights in conjunction with different angles to the part to be captured. For example, if the part is laying flat on a conveyor belt the lights would be placed at a 45-degree angle to it. This type of lighting is most useful when looking at highly reflective clear objects and is most commonly used with *lens scratch detections*.

The figure below shows the angular placement of light:

Field of view

In a vision workload, you need to know the distance to the object that you are trying to evaluate. This also will play a part in the camera selection, sensor selection, and lens configuration. Some of the components that make up the field of view are:

- **Distance to object(s):** For example, is the object being monitored with computer vision on a conveyor belt and the camera is two feet above it, or is the object across a parking lot? As the distance changes, so does the camera's sensors and lens configurations.
- **Area of coverage:** Is the area that the computer vision is trying to monitor small or large? This has direct correlation to the camera's resolution, lens, and sensor type.
- **Direction of the sun:** If the computer vision workload is outside, such as monitoring a job construction site for worker safety, will the camera be pointed in the sun at any time? Keep in mind that if the sun is casting a

shadow over the object that the vision workload is monitoring, items might be a bit obscured. Also, if the camera is getting direct sunlight in the lens, the camera might be *blinded* until the angle of the sun changes.

- **Camera angle to the object(s):** Angle of the camera to the object that the vision workload is monitoring is also a critical component to think about. If the camera is too high, it might miss the details that the vision workload is trying to capture, and the same may be true if it is too low.

Communication interface

In building a computer vision workload, it is also important to understand how the system will interact with the output of the camera. Below are a few of the standard ways that a camera will communicate to IoT Edge:

- **Real Time Streaming Protocol (RTSP):** RTSP is a protocol that transfers real-time video data from a device (in our case, the camera) to an endpoint device (Edge compute) directly over a TCP/IP connection. It functions in a client-server application model that is at the application level in the network.

- **Open Network Video Interface Forum (ONVIF):** A global and open industry forum that is developing open standards for IP-based cameras. This standard is aimed at standardization of communication between the IP camera and down stream systems, interoperability, and open source.

- **USB:** Unlike RTSP and ONVIF, USB connected cameras connect over the Universal Serial Bus directly on the Edge compute device. This is less complex, however, it limits the distance that the camera can be placed away from the Edge compute.

- **Camera Serial Interface:** CSI specification is from *Mobile Industry Processor Interface (MIPI)*. This interface describes how to communicate between a camera and a host processor.

  There are several standards defined for CSI:

  - **CSI-1**: This was the original standard that MIPI started with.
  - **CSI-2**: This standard was released in 2005, and uses either D-PHY or C-PHY as physical layers options. This is further divided into several layers:
    - Physical Layer (C-PHY, D-PHY)
    - Lane Merger layer
    - Low-Level Protocol Layer
    - Pixel to Byte Conversion Layer
    - Application layer

  The specification was updated in 2017 to v2, which added support for RAW-24 color depth, *Unified Serial Link*, and *Smart Region of Interest*.

Next steps

Now that you know the camera considerations for your IoT Edge Vision workload, proceed to setting up the right hardware for your workload. Read Hardware acceleration in Azure IoT Edge Vision for more information.

Hardware acceleration in Azure IoT Edge Vision

11/19/2021 • 2 minutes to read • Edit Online

Along with the camera selection, one of the other critical decisions in Vision on the Edge projects is hardware acceleration.

The following sections describe the key components of the underlying hardware.

CPU

The Central Processing Unit (CPU) is your default compute for most processes running on a computer. It is designed for general purpose compute. For some vision workloads where timing is not as critical, this might be a good option. However, most workloads that involve critical timing, multiple camera streams, and/or high frame rates, will require more specific hardware acceleration.

GPU

Many people are familiar with the Graphics Processing Unit (GPU) as this is the de-facto processor for any high-end PC graphics card. In recent years, the GPU has been leveraged in high performance computer (HPC) scenarios, in data mining, and in computer AI/ML workloads. The GPU's massive potential of parallel computing can be used in a vision workload to accelerate the processing of pixel data. The downside to a GPU is its higher power consumption, which is a critical factor to consider for your vision workload.

FPGA

Field Programmable Gate Arrays are reconfigurable hardware accelerators. These powerful accelerators allow for the growth of Deep Learning Neural networks, which are still evolving. These accelerators have millions of programmable gates, hundreds of I/O pins, and an exceptional compute power in the trillions of tera-MAC's. There also many different libraries available for FPGAs that are optimized for vision workloads. Some of these libraries also include preconfigured interfaces to connect to downstream cameras and devices. One area that FPGAs tend to fall short on is floating point operations. However, manufacturers are currently working on this issue and have made many improvements in this area.

ASIC

Application-Specific Integrated Circuit is by far the fastest accelerator on the market today. While they are the fastest, they are the hardest to change as they are manufactured to function for a specific task. These custom chips are gaining popularity due to size, power per watt performance, and IP protection. This is because the IP burned into the ASIC accelerator is much harder to backwards engineer proprietary algorithms.

Next steps

Proceed to learn what considerations go into place for Machine learning and data science in Azure IoT Edge Vision.

Machine learning and data science in Azure IoT Edge Vision

11/19/2021 • 12 minutes to read • Edit Online

The process of designing the machine learning (ML) approach for a vision on the edge scenario is one of the biggest challenges in the entire planning process. It is important to understand how to consider and think about ML in the context of edge devices.

To begin using machine learning to address business problems and pain points, consider the following points:

- Always consider first how to solve the problem without ML or with a simple ML algorithm.
- Have a plan to test several ML architectures as they will have different capacities to "learn".
- Have a system in place to collect new data from the device to retrain an ML model.
- For poorly performing ML models, often a simple fix is to add more representative data to the training process and ensure it has variability with all classes represented equally.
- Remember, this is often an iterative process with both the choice of data and choice of architecture being updated in the exploratory phase.

It is not an easy space and, for some, a very new way of thinking. It is a data driven process. Careful planning will be critical to successful results especially on very constrained devices.

It is always critical to clearly define the problem to be solved as the data science and machine learning approach will depend upon this. It is also very important to consider what type of data will be encountered in the edge scenario as this will determine the kind of ML algorithm that should be used.

Even at the start, before training any models, real world data collection and examination will help this process greatly and new ideas could even arise. This article will discuss data considerations in detail. Of course, the equipment itself will help determine the ML approach with regard to device attributes like limited memory, compute, and/or power consumption limits.

Fortunately, data science and machine learning are iterative processes, so if the ML model has poor performance, there are many ways to address issues through experimentation. This article will also discuss considerations around ML architecture choices. Often, there will be some trial and error involved as well.

Machine learning data

Both the source(s) and attributes of data will dictate how the intelligent edge system is built. For vision, it could be images, videos, or even LiDAR, as the streaming signal. Regardless of the signal, when training an ML model and using it to score new data (called *inferencing*), domain knowledge will be required. This includes experience in designing and using ML algorithms or neural network architectures and expertise deploying them to the specialized hardware. Below are a few considerations related to ML. However, it is recommended to gain some deeper knowledge in order to open up more possibilities or find an ML expert with edge experience to help with the project.

Collecting and using a *balanced dataset* is critical, and it should equally represent all classes or categories. When the ML model is trained on a dataset, generally that dataset has been split into train, validate, and test subsets. The purpose of these subsets is as follows:

- The training dataset is used for the actual model training over many passes or iterations (often called *epochs*).
- Throughout the training process, the model is spot-checked for how well it is doing on the validation dataset.

- After a model is done training, the final step is to pass the test dataset through it and assess how well it did as a proxy to the real-world.

> **NOTE**
> Be wary of optimizing for the test dataset, in addition to the training dataset, once one test has been run. It might be good to have a few different test datasets available.

Some good news is that in using deep learning, often costly and onerous feature engineering, featurizations, and preprocessing can be avoided because of how deep learning works to find signal in noise better than traditional ML. However, in deep learning, transformations may still be utilized to clean or reformat data for model input during training as well as inference. The same capacity needs to be used in training and when the model is scoring new data.

When advanced preprocessing is used such as denoising, adjusting brightness or contrast, or transformations like RGB to HSV, it must be noted that this can dramatically change the model performance for the better or, sometimes, for the worse. In general, it is part of the data science exploration process and sometimes it is something that must be observed once the device and other components are placed in a real-world location.

After the hardware is installed into its permanent location, the incoming data stream should be monitored for data drift.

Data drift is the deviation due to changes in the current data compared to the original. Data drift will often result in a degradation in model performance (like accuracy), although this is not the only cause of decreased performance (for example, hardware or camera failure).

There should be an allowance for data drift testing in the system. This new data should also be collected for another round of training. The more representative data collected for training, the better the model will perform in almost all cases. So, preparing for this kind of collection is always a good idea.

In addition to using data for training and inference, new data coming from the device could be used to monitor the device, camera or other components for hardware degradation.

In summary, here are the key considerations:

- Always use a balanced dataset with all classes represented equally.
- The more representative data used to train a model, the better.
- Have a system in place to collect new data from device to retrain.
- Have a system in place to test for data drift.
- Only run a test set through a new ML model once. If you iterate and retest on the same test set, this could cause overfitting to the test set in addition to the training set.

Machine learning architecture choices

Machine learning (ML) architecture is the layout of the mathematical operations that process input into the desired and actionable output. For instance, in deep learning this would be the number of layers and neurons in each layer of a deep neural network as well as their arrangement. It is important to note that there is no guarantee that the performance metric goal (such as, high enough accuracy) for any given ML architecture will be achieved. To mitigate this, several different architectures should be considered. Often, two or three different architectures are tried before a choice is made. Remember that this is often an iterative process; both the choice of data and the choice of architecture may be updated in the exploratory phase of the development process.

It helps to understand the issues that can arise when training an ML model that may only be seen after training or, even at the point of inferencing on device. Overfitting and underfitting are some of the common issues found during the training and testing process.

- **Overfitting:** Overfitting can give a false sense of success because the performance metric (like accuracy) might be very good when the input data looks like the training data. However, overfitting can occur when the model fits to the training data too closely and cannot generalize well to new data. For example, it may become apparent that the model only performs well indoors because the training data was from an indoor setting.

  Overfitting can be caused by following issues:

  - The model learned to focus on incorrect, non-representative features specifically found in the training dataset.
  - The model architecture may have too many learnable parameters, correlated to the number of layers in a neural network and units per layer. A model's *memorization capacity* is determined by the number of learnable parameters.
  - Not enough complexity or variation is found in the training data.
  - The model is trained over too many iterations.
  - There may be other reasons for good performance in training and significantly worse performance in validation and testing, which are out of scope for this article.
- **Underfitting:** Underfitting happens when the model has generalized so well that it cannot tell the difference between classes with confidence. For example, the training *loss* will still be unacceptably high.

  Underfitting can be caused by following issues:

  - Not enough samples available in training data.
  - The model is trained for too few iterations, in other words, it's too generalized.
  - Other reasons related to the model not being able to recognize any objects, or poor recognition and *loss values* during training. The assessment values used to direct the training process pass through a process called *optimization* and *weight updates*.

There is a trade-off between too much capacity (such as, a large network or a large number of learnable parameters) and too little capacity. *Transfer learning* happens when some network layers are set as not trainable, or *frozen*. In this situation, increasing capacity would equate to opening up more layers earlier in the network versus only using the last few layers in training, with the rest remaining frozen.

There is no hard and fast rule for determining number of layers for deep neural networks. So, sometimes several model architectures must be evaluated within an ML task. However, in general, it is good to start with fewer layers and/or parameters (such as, smaller networks) and gradually increase the complexity.

Some considerations when coming up with the best architecture choice will include the inference speed requirements. These include an assessment and acceptance of the speed versus accuracy tradeoff. Often, a faster inference speed is associated with lower performance. For example, accuracy, confidence or precision could suffer.

A discussion around requirements for the ML training and inferencing will be necessary based upon the considerations above and any company-specific requirements. For instance, if the company policy allows open-source solutions to be utilized, it will open up a great deal of ML algorithmic possibilities as most cutting edge ML work is in the open-source domain.

In summary, here are the key considerations:

- Keep an eye out for overfitting and underfitting.
- Testing several ML architectures is often a good idea. This is an iterative process.
- There will be a trade-off between too much network capacity and too little. However, it is better to start with too little and build up from there.
- There will be a trade-off between speed and your performance metric such as accuracy.
- If the performance of the ML model is acceptable, the exploratory phase is complete. This is important to

note, as one can be tempted to iterate indefinitely.

Data science workflows

The data science process for edge deployments has a general pattern. After a clear data-driven problem statement is formulated, the next steps generally include those shown in the figure below.

- **Data collection:** Data collection or acquisition could be an online image search from a currently deployed device, or other representative data source. Generally, the more data the better. In addition, the more variability, the better the generalization.
- **Data labeling:** If only hundreds of images need to be labeled, such as, when using transfer learning, it can be done in-house. If tens of thousands of images need to be labeled, a vendor could be enlisted for both data collection and labeling.
- **Train a model with ML framework:** An ML framework such as *TensorFlow* or *PyTorch* (both with Python and C++ APIs) will need to be chosen. Usually this depends upon what code samples are available in open-source or in-house, as well as the experience of the ML practitioner. Azure ML may be used to train a model using any ML framework and approach, as it is agnostic of framework and has Python and R bindings, and many wrappers around popular frameworks.
- **Convert the model for inferencing on device:** Almost always, a model will need to be converted to work with a particular runtime. Model conversion usually involves advantageous optimizations like faster

inference and smaller model footprints. This step differs for each ML framework and runtime. There are open-source interoperability frameworks available such as *ONNX* and *MMdnn*.

- **Build the solution for device:** The solution is usually built on the same type of device as will be used in the final deployment because binary files created system-specific.
- **Using runtime, deploy solution to device:** Once a runtime is chosen, usually in conjunction with the ML framework choice, the compiled solution may be deployed. The Azure IoT Runtime is a Docker-based system in which the ML runtimes may be deployed as containers.

The diagram below shows a sample data science process where open-source tools may be leveraged for the data science workflow. Data availability and type will drive most of the choices, including the devices/hardware chosen.

If a workflow already exists for the data scientists and app developers, a few other considerations may apply. First, it is advised to have a code, model, and data versioning system in place. Secondly, an automation plan for code and integration testing along with other aspects of the data science process, such as triggers, build/release process, and so on, will help speed up the time to production and cultivate collaboration within the team.

The language of choice can help dictate what API or SDK is used for inferencing and training ML models. This in

turn will then dictate what type of ML model, what type(s) of device, what type of IoT Edge module, and so on, need to be used. For example, PyTorch has a C++ API for inferencing and training, that works well in conjunction with the OpenCV C++ API. If the app developer working on the deployment strategy is building a C++ application, or has this experience, one might consider PyTorch or others (such as TensorFlow or CNTK) that have C++ inferencing APIs.

Machine learning and data science in a nutshell

In summary, here are the key considerations:

- Converting models also involves optimizations such as faster inference and smaller model footprints, critical for very resource-constrained devices.
- The solution will usually need to be built on a build-dedicated device, the same type of device to which the solution will be deployed.
- The language and framework of choice will depend upon both the ML practitioner's experience as well as what is available in open-source.
- The runtime of choice will depend upon the availability of the device and hardware acceleration for ML.
- It is important to have a code, model, and data versioning system.

Next steps

Proceed to Image storage and management in Azure IoT Edge Vision article to learn how to properly store the images created by your IoT Edge Vision solution.

Image storage and management in Azure IoT Edge Vision

11/19/2021 • 2 minutes to read • Edit Online

Storage and management of the images involved in a computer vision application is a critical function.

Some of the key considerations for managing these images are:

- Ability to store all raw images during training with ease of retrieval for labeling.
- Faster storage medium to avoid pipeline bottlenecks and loss.
- Storage on the edge as well as in the cloud, as labeling activity can be performed at both places.
- Categorization of images for easy retrieval.
- Naming and tagging images to link them with inferred metadata.

The combination of Azure Blob Storage, Azure IoT Hub, and Azure IoT Edge allow several potential options for the storage of image data, such as:

- Use of the Azure IoT Edge Blob Storage module, which will automatically sync images to Azure Blob based on policy.
- Storing images to local host file system and uploading to Azure Blob service using a custom module.
- Use of a local database to store images, which then are synced to the cloud database.

Typical storage workflow

The IoT Edge Blob Storage module is one of the most powerful and straightforward solutions, and our preferred approach. A typical workflow using this module might be as follows:

1. Raw messages after ingestion are stored locally on the Edge Blob module, with time stamping and sequence numbering to uniquely identify the image files.
2. Policy is set on the Edge Blob module for automatic upload to Azure Blob with ordering.
3. To conserve space on the Edge device, automatic deletion after certain time is configured along with *retain while uploading* option to ensure all images get synced to the cloud.
4. Local categorization or domain and labeling is implemented using module that can read these images into the UX. The label data is associated to the image URI along with the coordinates and category.
5. As label data needs to be saved, a local database is preferred to store this metadata, as it will allow easy lookup for the UX and can be synced to the cloud using telemetry messages.
6. During scoring run, the model detects matching patterns and generates events of interest. This metadata is sent to cloud via telemetry referring to the image URI and optionally stored in local database for edge UX. The images continue to be stored to Edge Blob and synced with Azure Blob.

Next steps

How you respond to alerts generated by the AI model is crucial. Learn more about this in Alert persistence in Azure IoT Edge Vision.

Alert persistence in Azure IoT Edge Vision

11/19/2021 • 2 minutes to read • Edit Online

In the context of vision on edge, an alert is a response to an event triggered by the AI model. In other words, it is the inferencing results. The type of event is determined by the training imparted to the model. These events are separate from operational events raised by the processing pipeline and any event related to the health of the runtime.

Types of alerts

Some of the common alerts types are:

- Image classification
- Movement detection
- Direction of movement
- Object detection
- Count of objects
- Total count of objects over period of time
- Average count of objects over period of time

Alerts are required to be monitored as they drive certain actions. They are critical to operations, being time sensitive in terms of processing, and are required to be logged for audit and further analysis.

Persistence of alerts

The alerts need to persist locally on the edge where they are raised and then passed on to the cloud for further processing and storage. This ensures a quick local response and avoids losing critical alerts due to any transient failures.

Some options to achieve this persistence and cloud syncing are:

- Utilize built-in store and forward capability of IoT Edge runtime, which automatically gets synced with Azure IoT Hub after a lost connectivity.
- Persist alerts on host file system as log files, which can be synced periodically to a blob storage in the cloud.
- Utilize Azure Blob Edge module, which will sync this data to Azure Blob in cloud based on policies that can be configured.
- Use local database on IoT Edge, such as SQL Edge for storing data, and sync with Azure SQL DB using SQL Data Sync. Another lightweight database option is the SQLite.

The preferred option is to use the built-in store and forward capability of IoT Edge runtime. This is more suitable for the alerts due to its time sensitivity, typically small messages sizes, and ease of use.

Next steps

Now you can proceed to work on the User interface in Azure IoT Edge Vision for your vision workload.

User interface in Azure IoT Edge Vision

11/19/2021 • 9 minutes to read • Edit Online

The user interface requirements of an IoT solution will vary depending on the overall objectives. Four types of user interfaces are commonly found in IoT solutions:

- **Administrator:** Allows full access to device provisioning, device and solution configuration, user management, and so on. These features could be provided as part of one solution or as separate solutions.
- **Consumer:** Is only applicable to consumer solutions. They provide similar access to the operator's interface, but limited to the devices owned by the user.
- **Operator:** Provides centralized access to the operational components of the solution. It typically includes device management, alerts monitoring, and configuration.
- **Analytics:** Is an interactive dashboard which provides visualization of telemetry and other data analyses.

This article focuses on a simple operator's user interface and visualization dashboard.

Technology options

- **Power BI:** Power BI is a compelling option for our analytics and virtualization needs. It provides power features to create customizable interactive dashboards. It also allows connectivity to many popular database systems and services. It is available as a managed service and as a self-hosted package. The former is the most popular and recommended option. With Power BI embedded in your solution, you could add customer-facing reports, dashboards, and analytics in your own applications by using and branding Power BI as your own. You can reduce developer resources by automating the monitoring, management, and deployment of analytics, while getting full control of Power BI features and intelligent analytics.

- **Azure Maps:** Another suitable technology for IoT visualizations is Azure Maps which allows you to create location-aware web and mobile applications using simple and secure geospatial services, APIs, and SDKs in Azure. You can deliver seamless experiences based on geospatial data with built-in location intelligence from world-class mobility technology partners.

- **Azure App Service:** Azure App Service is a managed platform with powerful capabilities for building web and mobile apps for many platforms and mobile devices. It allows developers to quickly build, deploy, and scale web apps created with popular frameworks, such as .NET, .NET Core, Node.js, Java, PHP, Ruby, or Python, in containers or running on any supported operating system. You can also meet rigorous, enterprise-grade performance, security, and compliance requirements by using the fully managed platform for your operational and monitoring tasks.

- **Azure SignalR Service:** For real-time data reporting, Azure SignalR Service, makes adding real-time communications to your web application as simple as provisioning a service. An in-depth real-time communications expertise is not required. It easily integrates with services such as Azure Functions, Azure Active Directory, Azure Storage, Azure App Service, Azure Analytics, Power BI, Azure IoT, Azure Cognitive Services, Azure Machine Learning, and more.

To secure your user interface solutions, **Azure Active Directory (Azure AD)** enterprise identity service provides single sign-on and multi-factor authentication.

Now let's learn how to build the user interface for some common scenarios.

Scenario 1

Contoso Boards produces high-quality circuit boards used in computers. Their number one product is a motherboard. Lately, they have been seeing an increase in issues with chip placement on the board. Through their investigation, they have noticed that the circuit boards are getting placed incorrectly on the assembly line. They need a way to identify if the circuit board is placed on the assembly line correctly. The data scientists at Contoso Boards are most familiar with TensorFlow and would like to continue using it as their primary ML model structure. Contoso Boards has several assembly lines that produce these mother boards. They would also like to centralize the management of the entire solution.

Considerations in this scenario

Contoso Boards can ask themselves questions such as the following:

- What are we analyzing?
  - Motherboard
- Where are we going to view the motherboard from?
  - Assembly Line Conveyor belt
- What camera do we need?
  - Area or line scan
  - Color or monochrome
  - CCD or CMOS sensor
  - Global or rolling shutter
  - Frame rate
  - Resolution
- What type of lighting is needed?
  - Backlighting
  - Shade
  - Darkfield
- How should the camera be mounted?
  - Top down
  - Side view
  - Angular
- What hardware should be used?
  - CPU
  - FPGA
  - GPU
  - ASIC

Solution

To find the solution that will be useful for Contoso Boards, let's focus on the edge detection. We need to **position a camera directly above at 90 degrees and about 16 inches above the edge part**. Since the conveyer system moves relatively slowly, we can use an **area scan** camera with a **global shutter**. For this use case, our camera should **capture about 30 frames per second**. The resolution can be found using the formula of `Res=(Object Size) Divided by (details to be captured)`. Based on this formula, **Res=16"/8"** gives 2MP in x and 4 in y, so we need a **camera capable of 4MP**. As for the sensor type, we are not fast moving, and really looking for an edge detection, so a **CMOS sensor** is better suited. One of the more critical aspects for any vision workload is lighting. In this application, Contoso Boards should choose to use a **white diffused filter back light**. This will make the part look almost black and have a high amount of contrast for edge detection. When it comes to color options for this application, it is better to be in black and white, as this is what will yield

the sharpest edge for the AI detection model. The data scientists are most familiar with TensorFlow, so learning ONNX or others would slow down the time for development of the model. Also because there are several assembly lines that will use this solution, and Contoso Boards would like a centrally managed edge solution, so **Azure Stack Edge** (with GPU option) would work well here. Based on the workload, the fact that Contoso Boards already knows TensorFlow, and this will be used on multiple assembly lines, GPU-based hardware would be the choice for hardware acceleration.

The following figure shows a sample of what the camera would see in this scenario:

Scenario 2

Contoso Shipping recently has had several pedestrian accidents at their loading docks. Most of the accidents are happening when a truck leaves the loading dock, and the driver does not see a dock worker walking in front of the truck. Contoso Shipping would like a solution that would watch for people, predict the direction of travel, and warn the drivers of potential dangers of hitting the workers. The distance from the cameras to Contoso Shipping's server room is too far for GigE connectivity, however they do have a large WIFI mesh that could be used. Most of the data scientists at Contoso Shipping are familiar with Open-VINO and they would like to be able to reuse the models on additional hardware in the future. The solution will also need to ensure that devices are operating as power-efficiently as possible. Finally, Contoso Shipping needs a way to manage the solution remotely for updates.

Considerations in this scenario

Contoso Shipping can introspect by asking the following questions:

- What are we analyzing?
  - People and patterns of movement
- Where are we going to view the people from?
  - The loading docks are 165 feet long.
  - Cameras will be placed 17 feet high to keep with city ordinances.
  - Cameras will need to be positioned 100 feet away from the front of the trucks.
  - Camera focus will need to be 10 feet behind the front of the truck, and 10 additional feet in front of the truck, giving a 20 foot depth on focus.
- What camera do we need?
  - Area or line scan
  - Color or monochrome
  - CCD or CMOS sensor
  - Global or rolling shutter
  - Frame rate
  - Resolution
- What type of lighting is needed?
  - Backlighting
  - Shade
  - Darkfield
- What hardware should be used?
  - CPU
  - FPGA
  - GPU
  - ASIC
- How should the camera be mounted?
  - Top down
  - Side view
  - Angular

Solution

Based on the distance of the loading dock size, Contoso Shipping will require several cameras to cover the entire dock. The zoning laws that Contoso Shipping must adhere to, require that the surveillance cameras cannot be mounted higher that 20 feet. In this use case, the average size of a worker is 5 foot 8 inches. The solution must use the least number of cameras possible.

Formula for field of view (FOV):

$$\frac{\text{Horizontal Resolution}}{\text{Pixels per Foot}} = \text{Field of View}$$

For example, look at the following images.

This image is taken with 480 horizontal pixels at 20 foot:

This image is taken with 5184 horizontal pixels at 20 foot:

The red square is shown to illustrate one pixel color.

These images demonstrate the issue with using the wrong resolution camera for a given use case. Lens can impact the FOV. However, if the wrong sensor is used for that given use case, the results could be less than expected.

With the above in mind, when choosing a camera for the solution required for Contoso Shipping, we need to think about how many cameras, and at what resolution, are needed to get the correct number of details to detect a person. Since we are only trying to identify if a person is in the frame or not, our PPF does not need to be around 80, which is needed for facial identification; we can use somewhere around 15-20. That would place the FOV around 16 feet. A 16-foot FOV would give us about 17.5 pixels per foot, which fits within our required PPF of 15-20. This would mean that we need a **10MP camera that has a horizontal resolution of ~5184**

pixels, and a lens that would allow for an **FOV of 16 feet**. The cameras would need to be placed outside, and the choice of sensor type should not allow for *bloom*. Bloom is when light hits the sensor and overloads the sensor, causing a view of almost over-exposure or a *white out* kind of condition. **CMOS** is the sensor of choice here.

Contoso operates 24x7 and as such, needs to ensure that nighttime personnel are also protected. **Monochrome** handles low light conditions much better compared to color. We are not looking to identify a person based on color. And monochrome sensors are a little cheaper then color.

How many cameras will it take? Since we have figured out that our cameras can look at a 16 foot path, we can do simple math. 165 foot dock divided by 16 foot FOV gives us 10.3125 cameras. So the solution would need **11 monochrome 5184 horizontal pixel (or 10MP) CMOS cameras with IPX67 housings or weather boxes**. The cameras would be mounted on 11 poles 100 feet from the trucks at 17f high. Based on the fact that the data scientists are more familiar with **Open-VINO**, data models should be built in **ONNX**. As for what hardware should be used, a device that can be connected over WIFI, and use as little power as possible, is required. Based on this, they should look to an **FPGA processor**. Potentially an ASIC processor could also be utilized, but due to the nature of how an ASIC processor works, it would not meet the requirement of being able to use the models on different hardware in the future.

Next steps

This series of articles have demonstrated how to build a complete vision workload using Azure IoT Edge devices. For further information, you may refer to the product documentation as following:

- Azure IoT Edge documentation
- Tutorial: Perform image classification at the edge with Custom Vision Service
- Azure Machine Learning documentation
- Azure Kinect DK documentation
- MMdnn tool
- ONNX

Azure industrial IoT analytics guidance

11/19/2021 • 11 minutes to read • Edit Online

This article series shows a recommended architecture for an Industrial IoT (IIoT) *analytics solution* on Azure using PaaS (Platform as a service) components. Industrial IoT or IIoT is the application of *Internet of Things* in the manufacturing industry.

An IIoT analytics solution can be used to build a variety of applications that provide:

- Asset monitoring
- Process dashboards
- Overall equipment effectiveness (OEE)
- Predictive maintenance
- Forecasting

Such an IIoT analytics solution relies on real-time and historical data from industrial devices and control systems located in discrete and process manufacturing facilities. These include PLCs (Programmable Logic Controller), industrial equipment, SCADA (Supervisory Control and Data Acquisition) systems, MES (Manufacturing Execution System), and Process Historians. The architecture covered by this series, includes guidance for connecting to all these systems.

A modern IIoT analytics solution goes beyond moving existing industrial processes and tools to the cloud. It involves transforming your operations and processes, embracing PaaS services, and leveraging the power of machine learning and the intelligent edge to optimize industrial processes.

The following list shows some typical *personas* who would use the solution and how they would use this solution:

- **Plant Manager:** responsible for the entire operations, production, and administrative tasks of the manufacturing plant.
- **Production Manager:** responsible for production of a certain number of components.
- **Process Engineer:** responsible for designing, implementing, controlling, and optimizing industrial processes.
- **Operations Manager:** responsible for overall efficiency of operation in terms of cost reduction, process time, process improvement, and so on.
- **Data Scientist:** responsible for building and training predictive Machine Learning models using historical industrial telemetry.

The following architecture diagram shows the core subsystems that form an IIoT analytics solution.

> **NOTE**
>
> This architecture represents an ingestion-only pattern. No control commands are sent back to the industrial systems or devices.

The architecture consists of a number of subsystems and services, and makes use of the Azure Industrial IoT components. Your own solution may not use all these services or may have additional services. This architecture also lists alternative service options, where applicable.

> **IMPORTANT**
>
> This architecture includes some services marked as "Preview" or "Public Preview". Preview services are governed by Supplemental Terms of Use for Microsoft Azure Previews.

Industrial systems and devices

In process manufacturing, industrial equipment (for example, flow monitors, pumps, and so on) is often geographically dispersed and must be monitored remotely. Remote Terminal Units (RTUs) connect remote equipment to a central SCADA system. RTUs work well in conditions where connectivity is intermittent, and no reliable continuous power supply exists.

In discrete manufacturing, industrial equipment (for example, factory robots, conveyor systems, and so on) are connected and controlled by a PLC. One or more PLCs may be connected to a central SCADA system using protocols such as Modbus or other industrial protocols.

In some cases, the data from SCADA systems is forwarded and centralized in an MES or a *historian* software (also known as *process historian* or *operational historian*). These historians are often located in IT controlled networks and have some access to the internet.

Frequently, industrial equipment and SCADA systems are in a closed Process Control Network (PCN), behind one or more firewalls, with no direct access to the internet. Historians often contain industrial data from multiple facilities and are located outside of a PCN. So, connecting to a historian is often the path of least resistance, rather than connecting to a SCADA, MES, or PLC. If a historian is not available, then connecting to an MES or SCADA system would be the next logical choice.

The connection to the historian, MES, or SCADA system will depend on what protocols are available on that system. Many systems now include Industry 4.0 standards such as OPC UA. Older systems may only support legacy protocols such as Modbus, ODBC, or SOAP. If so, you will most likely require a protocol translation software running on an *intelligent edge* device.

Intelligent edge

Intelligent edge devices perform some data processing on the device itself or on a field gateway. In most industrial scenarios, the industrial equipment cannot have additional software installed on it. This is why a field gateway is required to connect the industrial equipment to the cloud.

Azure IoT Edge

To connect industrial equipment and systems to the cloud, we recommend using Azure IoT Edge as the field gateway for:

- Protocol and identity translation;
- Edge processing and analytics; and
- Adhering to network security policies (ISA 95, ISA 99).

Azure IoT Edge is a free, open source field gateway software that runs on a variety of supported hardware devices or a virtual machine.

IoT Edge allows you to run edge workloads as Docker container modules. The modules can be developed in several languages, with SDKs provided for Python, Node.js, C#, Java and C. Prebuilt Azure IoT Edge modules from Microsoft and third-party partners are available from the Azure IoT Edge Marketplace.

Real-time industrial data is encrypted and streamed through Azure IoT Edge to Azure IoT Hub using AMQP 1.0 or MQTT 3.1.1 protocols. IoT Edge can operate in offline or intermittent network conditions providing ***store and forward*** capabilities.

There are two system modules provided as part of IoT Edge runtime.

- The **EdgeAgent** module is responsible for pulling down the container orchestration specification (manifest) from the cloud, so that it knows which modules to run. Module configuration is provided as part of the module twin.
- The **EdgeHub** module manages the communication from the device to Azure IoT Hub, as well as the inter-module communication. Messages are routed from one module to the next using JSON configuration.

Azure IoT Edge automatic deployments can be used to specify a standing configuration for new or existing devices. This provides a single location for deployment configuration across thousands of Azure IoT Edge devices.

A number of third-party IoT Edge gateway devices are available from the *Azure Certified for IoT Device Catalog*.

> **IMPORTANT**
> Proper hardware sizing of an IoT Edge gateway is important to ensure edge module performance. See the performance considerations for this architecture.

Gateway patterns

There are three patterns for connecting your devices to Azure via an IoT Edge field gateway (or virtual machine):

1. **Transparent:** Devices already have the capability to send messages to IoT Hub using AMQP or MQTT. Instead of sending the messages directly to the hub, they instead send the messages to IoT Edge, which in turn passes them on to IoT Hub. Each device has an identity and device twin in Azure IoT Hub.

2. **Protocol Translation:** Also known as an opaque gateway pattern. This pattern is often used to connect older brownfield equipment (for example, Modbus) to Azure. Modules are deployed to Azure IoT Edge to perform the protocol conversion. Devices must provide a unique identifier to the gateway.

3. **Identity Translation:** In this pattern, devices cannot communicate directly to IoT Hub (for example, OPC UA Pub/Sub, BLE devices). The gateway is smart enough to understand the protocol used by the downstream devices, provide them identity, and translate IoT Hub primitives. Each device has an identity and device twin in Azure IoT Hub.

Although you can use any of these patterns in your IIoT Analytics Solution, your choice will be driven by which protocol is installed on your industrial systems. For example, if your SCADA system supports ethernet/IP, you will need to use a protocol translation software to convert ethernet/IP to MQTT or AMQP. See the Connecting to Historians section for additional guidance.

IoT Edge gateways can be provisioned at scale using the Azure IoT Hub Device Provisioning Service (DPS). DPS is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention, enabling customers to provision millions of devices in a secure and scalable manner.

OPC UA

OPC UA is the successor to OPC Classic (OPC DA, AE, HDA). The OPC UA standard is maintained by the OPC Foundation. Microsoft has been a member of the OPC Foundation since 1996 and has supported OPC UA on Azure since 2016.

Industry and domain-specific *Information Models* can be created based on the OPC UA *Data Model*. The specifications of such Information Models (also called *industry standard models* since they typically address a dedicated industry problem) are called Companion Specifications. The synergy of the OPC UA infrastructure to exchange such industry information models enables interoperability at the semantic level. OPC UA can use a number of transport protocols including MQTT, AMQP, and UADP.

Microsoft has developed open source Azure Industrial IoT components, based on OPC UA, which implement identity translation pattern:

- OPC Twin consists of microservices and an Azure IoT Edge module to connect the cloud and the factory network. OPC Twin provides discovery, registration, and synchronous remote control of industrial devices through REST APIs.
- OPC Publisher is an Azure IoT Edge module that connects to existing OPC UA servers and publishes telemetry data from OPC UA servers in OPC UA PubSub format, in both JSON and binary.
- OPC Vault is a microservice that can configure, register, and manage certificate lifecycle for OPC UA server and client applications in the cloud.
- Discovery Services is an Azure IoT Edge module that supports network scanning and OPC UA discovery.

The Microsoft Azure IIoT solution also contains a number of services, REST APIs, deployment scripts, and configuration tools that you can integrate into your IIoT analytics solution. These are open source and available on GitHub.

Edge workloads

The ability to run custom or third-party modules at the edge is important.

- If you want to respond to emergencies as quickly as possible, you can run anomaly detection or Machine Learning module in tight control loops at the edge.
- If you want to reduce bandwidth costs and avoid transferring terabytes of raw data, you can clean and aggregate the data locally then send only the insights to the cloud for analysis.
- If you want to convert legacy industrial protocols, you can develop a custom module or purchase a third-party module for protocol translation.
- If you want to quickly respond to an event on the factory floor, you can use an edge module to detect that event and another module to respond to it.

Microsoft and our partners have made available on the Azure Marketplace a number of edge modules, which can be used in your IIoT analytics solution. Protocol and identity translation are the most common edge workloads used within an IIoT analytics solution. In the future, expect to see other workloads such as closed loop control using edge ML models.

Connecting to historians

A common pattern when developing an IIoT analytics solution is to connect to a process historian and stream real-time data from the historian to Azure IoT Hub. How this is done, will depend on which protocols are installed and accessible (that is, not blocked by firewalls) on the historian.

PROTOCOL AVAILABLE ON HISTORIAN	OPTIONS
OPC UA	- Use Azure IoT Edge, along with OPC Publisher, OPC Twin, and OPC Vault, to send OPC UA data over MQTT to IoT Hub. OPC Twin also has support for OPC UA HDA profile, useful for obtaining historical data. - Use a third-party Azure IoT Edge OPC UA module to send OPC UA data over MQTT to IoT Hub.
OPC DA	- Use third-party software to convert OPC DA to OPC UA and send OPC UA data to IoT Hub over MQTT. Or use OPC Publisher, OPC Twin and OPC Vault to send OPC UA data over MQTT to IoT Hub.
Web Service	- Use a custom Azure IoT Edge HTTP module to poll the web service. - Use third-party software that supports HTTP to MQTT 3.1.1 or AMQP 1.0.
MQTT 3.1.1 (Can publish MQTT messages)	- Connect historian directly to Azure IoT Hub using MQTT. - Connect historian to Azure IoT Edge as a leaf device. See Transparent Gateway pattern.

PROTOCOL AVAILABLE ON HISTORIAN	OPTIONS
Other	- Use a custom Azure IoT Edge module. - Use third-party software to convert to MQTT 3.1.1 or AMQP 1.0.

A number of Microsoft partners have developed protocol and identity translation modules or solutions that are available on the Azure Marketplace.

Some historian vendors also provide first-class capabilities to send data to Azure.

HISTORIAN	OPTIONS
OSIsoft PI	PI Integrator for Azure
Honeywell	Uniformance Cloud Historian

Once real time data streaming has been established between your historian and Azure IoT Hub, it is important to export your historian's historical data and import it into your IIoT analytics solution. For guidance on how to accomplish this, see Historical Data Ingestion.

Cloud gateway

A cloud gateway provides a cloud hub for devices and field gateways to connect securely to the cloud and send data. It also provides device management capabilities. For the cloud gateway, we recommend Azure IoT Hub. IoT Hub is a hosted cloud service that ingests events from devices and IoT Edge gateways. IoT Hub provides secure connectivity, event ingestion, bidirectional communication, and device management. When IoT Hub is combined with the Azure Industrial IoT components, you can control your industrial devices using cloud-based REST APIs.

IoT Hub supports the following protocols:

- MQTT 3.1.1,
- MQTT over WebSockets,
- AMQP 1.0,
- AMQP over WebSockets, and
- HTTPS.

If the industrial device or system supports any of these protocols, it can send data directly to IoT Hub. In most industrial environments, this is not permissible because of PCN firewalls and network security policies (ISA 95, ISA 99). In such cases, an Azure IoT Edge field gateway can be installed in a DMZ between the PCN and the Internet.

Next steps

To learn the services recommended for this architecture, continue reading the series with Services in an IIoT analytics solution.

Services in an Industrial IoT analytics solution

11/19/2021 • 19 minutes to read • Edit Online

Building on the architectural components in the recommended Azure Industrial IoT analytics solution, this article discusses the subsystems and Azure services that can be used in such a solution. Your solution may not use all these services or may have additional services.

Time Series Service

A time series service will be required to provide a warm and cold store for the JSON-based industrial time series data. We recommend using Time Series Insights (or TSI) as the time series service for the following reasons:

- Ability to ingest streaming data directly from Azure IoT Hub or Azure Event Hub.
- Multi-layered storage solution with warm and cold analytics support. It provides you the option to route data between warm and cold, for interactive analytics over warm data as well as operational intelligence over decades of historical data.
  - A highly interactive warm analytics solution to perform frequent, and large number of queries over shorter time span data.
  - A scalable, performant, and cost-optimized time series data lake based on Azure Storage allowing customers to trend years' worth of time series data.
- Asset hierarchy support that describes the domain and metadata associated with the derived and raw signals from industrial assets and devices. Multiple hierarchies can be defined reflecting different departments within your company. For instance, a factory hierarchy will look different from an Oil & Gas hierarchy.
- Flexible analytics platform to store historical time series data in your own Azure Storage account, thereby allowing you to have ownership of your industrial IoT data. Data is stored in open source Apache Parquet format that enables connectivity and interoperability across a variety of data scenarios including predictive analytics, machine learning, and other custom computations done using familiar technologies including Spark, Databricks, and Jupyter.
- Rich analytics with enhanced query APIs and user experience that combines asset-based data insights with rich, improvised data analytics with support for interpolation, scalar and aggregate functions, categorical variables, scatter plots, and time-shifting time series signals for in-depth analysis.
- Enterprise grade platform to support the scale, performance, security, and reliability needs of our industrial IoT customers.
- Ability to view and export the data in CSV format for further analysis.
- Ability to share analysis across your organization. Current visualizations can be saved and shared. Alternatively, the current state of each user's analysis is assigned a unique identifier. This identifier is placed in the URL allowing users to easily share their analysis by placing the URL in emails, documents, etc. Because TSI pricing is not *seat*-based, this democratizes the data by allowing anyone with access to the data to see it.

You can continue to use Time Series Insights with the old pricing model (S1, S2) for warm, unplanned data analysis. However, we highly encourage you to use the updated offering (PAYG) as it offers many new capabilities as well as Pay-As-You-Go pricing, cost savings, and flexibility to scale. Alternatively, Azure Data Explorer or Cosmos DB may be used as a time series database if you plan to develop a custom time series

service.

> **NOTE**
> When connecting Time Series Insights with IoT Hub or Event Hub, ensure you select an appropriate Time Series ID. We recommend using a SCADA tag name field or OPC UA node id (for example, nsu=http://msft/boiler;i=#####) if possible, as these will map to leaf nodes in your Time Series Model.

The data in Time Series Insights is stored in your Azure Blob Storage account (bring your own storage account) in Parquet file format. It is your data after all!

You can query your data in Time Series Insights using:

- Time Series Insights Explorer
- Query API
- REST API
- Power BI
- Any of your favorite BI and analytics tools (for example, Spark, Databricks, Azure Notebooks) by accessing the Parquet files in your Azure blob storage account

Microservices

Your IIoT analytics solution will require a number of microservices to perform functions such as:

- Providing HTTP REST APIs to support your web application.
  - We recommend creating HTTP-triggered Azure Functions to implement your APIs.
  - Alternatively, you can develop and host your REST APIs using Azure Service Fabric or Azure Kubernetes Service (AKS).
- Providing an HTTP REST API interface to your factory floor OPC UA servers (for example, using Azure Industrial IoT components consisting of OPC Publisher, OPC Twin and OPC Vault) to provide discovery, registration, and remote control of industrial devices.
  - For hosting the Azure Industrial IoT microservices, we recommend using Azure Kubernetes Service (AKS). See Deploying Azure Industrial IoT Platform to understand the various deployment options.
- Performing data transformation such as converting binary payloads to JSON or differing JSON payloads to a common, canonical format.
  - We recommend creating Azure Functions connected to IoT Hub to perform payload transformations.
  - Different industrial equipment vendors will send telemetry in different payload formats (JSON, binary, and so on) and schemas. When possible, we recommend converting the different equipment schemas to a common, canonical schema, ideally based on an industry standard.
  - If the message body is binary, use an Azure Function to convert the incoming messages to JSON and send the converted messages back to IoT Hub or to Event Hub.
    - When the message body is binary, IoT Hub message routing cannot be used against the message body, but can be used against message properties.
    - The Azure Industrial IoT components include the capability to decode OPC UA binary messages to JSON.
- A Data Ingest Administration service for updating the list of tags monitored by your IIoT analytics solution.

- A Historical Data Ingestion service for importing historical data from your SCADA, MES, or historian into your solution.

Your solution will likely involve additional microservices to satisfy the specific requirements of your IIoT analytics solution. If your organization is new to building microservices, we recommend implementing custom microservices using Azure Functions. Azure Functions is an event-driven serverless compute platform that can be used to develop microservices and solve complex orchestration problems. It allows you to build and debug locally (in several software languages) without additional setup, deploy and operate at scale in the cloud, and integrate Azure services using triggers and bindings.

Both stateless and stateful microservices can be developed using Azure Functions. Azure Functions can use Cosmos DB, Table Storage, Azure SQL, and other databases to store stateful information.

Alternatively, if your organization has a previous experience building container-based microservices, we recommend you to also consider Azure Service Fabric or Azure Kubernetes Service (AKS). Refer to Microservices in Azure for more information.

Regardless of your microservices platform choice, we recommend using Azure API Management to create consistent and modern API gateways for your microservices. API Management helps abstract, publish, secure, and version your APIs.

Data Ingest Administration

We recommend developing a Data Ingest Administration service to add/update the list of tags monitored by your IIoT analytics solution.

SCADA *tags* are variables mapped to I/O addresses on a PLC or RTU. Tag names vary from organization to organization but often follow a naming pattern. As an example, tag names for a pump with tag number `14P103` located in STN001 (Station 001), has these statuses:

- STN001_14P103_RUN
- STN001_14P103_STOP
- STN001_14P103_TRIP

As new tags are created in your SCADA system, the IIoT analytics solution must become aware of these tags and subscribe to them in order to begin collecting data from them. In some cases, the IIoT analytics solution may not subscribe to certain tags as the data they contain may be irrelevant.

If your SCADA system supports OPC UA, new tags should appear as new NodeIDs in the OPC UA hierarchy. For example, the above tag names may appear as:

- ns=2;s= STN001_14P103_RUN
- ns=2;s= STN001_14P103_STOP
- ns=2;s= STN001_14P103_TRIP

We recommend developing a workflow that informs the administrators of the IIoT analytics solution when new tags are created, or existing tags are edited in the SCADA system. At the end of the workflow, the OPC Publisher is updated with the new/updated tags.

To accomplish this, we recommend developing a workflow that involves Power Apps, Logic Apps, and Azure Functions, as follows:

- The SCADA system operator can trigger the Logic Apps workflow using a Power Apps form whenever tags are created or edited in the SCADA system.
  - Alternatively, Logic Apps connectors can monitor a table in the SCADA system database for tag changes.
  - The OPC UA Discovery service can be used to both find OPC UA servers and the tags and methods

they implement.

- The Logic Apps workflow includes an approval step where the IIoT analytics solution owners can approve the new/updated tags.
- Once the new/updated tags are approved and a frequency assigned, the Logic App calls an Azure Function.
- The Azure function calls the OPC Twin microservice, which directs the OPC Publisher module to subscribe to the new tags.
  - A sample can be found here.
  - If your solution involves third-party software, instead of OPC Publisher, configure the Azure Function to call an API running on the third-party software either directly or using an IoT Hub Direct Method.

Alternatively, Microsoft Forms and Microsoft Flow can be used in place of Power Apps and Logic Apps.

Historical Data Ingestion

Years of historical data likely exists in your current SCADA, MES, or historian system. In most cases, you will want to import your historical data into your IIoT analytics solution.

Loading historical data into your IIoT analytics solution consists of three steps:

1. Export your historical data.
   a. Most SCADA, MES, or historian systems have some mechanism that allows you to export your historical data, often as CSV files. Consult your system's documentation on how best to do this.
   b. If there is no export option in your system, consult the system's documentation to determine if an API exists. Some systems support HTTP REST APIs or OPC Historical Data Access (HDA). If so, build an application or use a Microsoft partner solution that connects to the API, queries for the historical data, and saves it to a file in formats such as CSV, Parquet, TSV, and so on.
2. Upload to Azure.
   a. If the aggregate size of the exported data is small, you can upload the files to Azure Blob Storage over the internet using Azcopy.
   b. If the aggregate size of the exported data is large (tens or hundreds of TBs), consider using Azure Import/Export Service or Azure Data Box to ship the files to the Azure region where your IIoT analytics solution is deployed. Once received, the files will be imported into your Azure Storage account.
3. Import your data.
   a. This step involves reading the files in your Azure Storage account, serializing the data as JSON, and sending data as streaming events into Time Series Insights. We recommend using an Azure Function to perform this.
   b. Time Series Insights only supports IoT Hub and Event Hub as data sources. We recommend using an Azure Function to send the events to a temporary Event Hub, which is connected to Time Series Insights.
   c. Refer to How to shape JSON events and Supported JSON shapes for best practices on shaping your JSON payload.
   d. Make sure to use the same Time Series ID as you do for your streaming data.
   e. Once this process is completed, the Event Hub and Azure Function may be deleted. This is an optional step.

> **NOTE**
> Exporting large volumes of data from your industrial system (for example, SCADA or historian) may place a significant performance load on that system, which can negatively impact operations. Consider exporting smaller batches of historical data to minimize performance impacts.

Rules and Calculation Engine

Your IIoT analytics solution may need to perform near real-time (low latency) calculations and complex event processing (or CEP) over streaming data, before it lands in a database. For example, calculating moving averages or calculated *tags*. This is often referred to as a *calculations engine*. Your solution may also need to trigger actions (for example, display an alert) based on the streaming data. This is referred to as a *rules engine*.

We recommend using Time Series Insights for simple calculations, at query time. The Time Series Model introduced with Time Series Insights supports a number of formulas including: Avg, Min, Max, Sum, Count, First, and Last. The formulas can be created and applied using the Time Series Insights APIs or Time Series Insights Explorer user interface.

For example, a Production Manager may want to calculate the average number of widgets produced on a manufacturing line, over a time interval, to ensure productivity goals are met. In this example, we would recommend the Production Manager to use the Time Series Insights explorer interface to create and visualize the calculation. Or if you have developed a custom web application, it can use the Time Series Insights APIs to create the calculation, and the Azure Time Series Insights JavaScript SDK (or *tsiclient*) to display the data in your custom web application.

For more advanced calculations and/or to implement a rules engine, we recommend using Azure Stream Analytics. Azure Stream Analytics is a real-time analytics and complex event-processing engine, that is designed to analyze and process high volumes of fast streaming data from multiple sources simultaneously. Patterns and relationships can be identified in information extracted from a number of input sources including devices, sensors, click streams, social media feeds, and applications. These patterns can be used to trigger actions and initiate workflows such creating alerts, feeding information to a reporting tool, or storing transformed data for later use.

For example, a Process Engineer may want to implement a more complex calculation such as calculating the standard deviation (SDEV) of the widgets produced across a number of production lines to determine when any line is more than 2x beyond the mean over a period of time. In this example, we recommend using Stream Analytics, with a custom web application. The Process Engineer authors the calculations using the custom web application, which calls the Stream Analytics REST APIs to create and run these calculations (also known as *Jobs*). The Job output can be sent to an Event Hub, connected to Time Series Insights, so the result can be visualized in Time Series Insights explorer.

Similarly, for a *Rules Engine*, a custom web application can be developed that allows users to author alerts and actions. The web application creates associated Jobs in Azure Stream Analytics using the Steam Analytics REST API. To trigger actions, a Stream Analytics Job calls an Azure Function output. The Azure Function can call a Logic App or Power Automate task that sends an Email alert or invokes Azure SignalR to display a message in the web application.

Azure Stream Analytics supports processing events in CSV, JSON, and Avro data formats while Time Series Insights supports JSON. If your payload does not meet these requirements, consider using an Azure Function to perform data transformation prior to sending the data to Stream Analytics or Time Series Insights (using IoT Hub or Event Hubs).

Azure Stream Analytics also supports reference data, a finite data set that is static or slowly changing in nature, used to perform a lookup or to augment your data streams. A common scenario is exporting asset metadata from your Enterprise Asset Management system and joining it with real-time data coming from those industrial devices.

Stream Analytics is also available as a module on the Azure IoT Edge runtime. This is useful for situations where complex event processing needs to happen at the Edge. As an alternative to Azure Stream Analytics, near real-time Calculation and Rules Engines may be implemented using Apache Spark Streaming on Azure Databricks.

Notifications

Since the IIoT analytics solution is *not a control system*, it does not require a complete Alarm Management system. However, there will be cases where you will want the ability to detect conditions in the streaming data and generate notifications or trigger workflows. Examples include:

- temperature of a heat exchanger exceeding a configured limit, which changes the color of an icon in your web application,
- an error code sent from a pump, which triggers a work order in your ERP system, or
- the vibration of a motor exceeding limits, which triggers an email notification to an Operations Manager.

We recommend using Azure Stream Analytics to define and detect conditions in the streaming data (refer to the rules engine mentioned earlier). For example, a Plant Manager implements an automated workflow that runs whenever an error code is received from any equipment. In this example, your custom web application can use the Stream Analytics REST API to provide a user interface for the Plant Manager to create and run a job that monitors for specific error codes.

For defining an alert (email or SMS) or triggering a workflow, we recommend using Azure Logic Apps. Logic Apps can be used to build automated, scalable workflows, business processes, and enterprise orchestrations to integrate your equipment and data across cloud services and on-premises systems.

We recommend connecting Azure Stream Analytics with Azure Logic Apps using Azure Service Bus. In the previous example, when an error code is detected by Stream Analytics, the job will send the error code to an Azure Service Bus queue output. A Logic App will be triggered to run whenever a message is received on the queue. This Logic App will then perform the workflow defined by the Plant Manager, which may involve creating a work order in Dynamics 365 or SAP, or sending an email to maintenance technician. Your web application can use the Logic Apps REST API to provide a user interface for the Plant Manager to author workflows or these can be built using the Azure portal authoring experience.

To display visual alerts in your web application, we recommend creating an Azure Stream Analytics job to detect specific events and send those to either:

- **An Event Hub output:** Then connect the Event Hub to Time Series Insights. Use the Azure Time Series Insights JavaScript SDK (tsiclient) to display the event in your web application.

or,

- **An Azure Functions output:** Then develop an Azure Function that sends the events to your web application using SignalR.

Operational alarms and events triggered on premise can also be ingested into Azure for reporting and to trigger work orders, SMS messages, and emails.

Microsoft 365

The IIoT analytics solution can also include Microsoft 365 services to automate tasks and send notifications. The following are a few examples:

- Receive email alerts in Microsoft Outlook or post a message to a Microsoft Teams channel when a condition is met in Azure Stream Analytics.
- Receive notifications as part of an approval workflow triggered by a Power App or Microsoft Forms submission.
- Create an item in a SharePoint list when an alert is triggered by a Logic App.
- Notify a user or execute a workflow when a new tag is created in a SCADA system.

Machine Learning

Machine learning models can be trained using your historical industrial data, enabling you to add predictive capabilities to your IIoT application. For example, your Data Scientists may be interested in using the IIoT analytics solution to build and train models that can predict events on the factory floor or indicate when maintenance should be conducted on an asset.

For building and training machine learning models, we recommend Azure Machine Learning. Azure Machine Learning can connect to Time Series Insights data stored in your Azure Storage account. Using the data, you can create and train forecasting models in Azure Machine Learning. Once a model has been trained, it can be deployed as a web service on Azure (hosted on Azure Kubernetes Services or Azure Functions, for example) or to an Azure IoT Edge field gateway.

For those new to machine learning or organizations without Data Scientists, we recommend starting with Azure Cognitive Services. Azure Cognitive Services are APIs, SDKs, and services available to help you build intelligent applications without having formal AI or data science skills or knowledge. Azure Cognitive Services enable you to easily add cognitive features into your IIoT analytics solution. The goal of Azure Cognitive Services is to help you create applications that can see, hear, speak, understand, and even begin to reason. The catalog of services within Azure Cognitive Services can be categorized into five main pillars - *Vision*, *Speech*, *Language*, *Web Search*, and *Decision*.

Asset Hierarchy

An asset hierarchy allows you to define hierarchies for classifying your asset, for example, Country > Location > Facility > Room. They may also contain the relationship between your assets. Many organizations maintain asset hierarchies within their industrial systems or within an Enterprise Asset Management (EAM) system.

The Time Series Model in Azure Time Series Insights provides asset hierarchy capabilities. Through the use of *Instances*, *Types* and *Hierarchies*, you can store metadata about your industrial devices, as shown in the image below.

If possible, we recommend exporting your existing asset hierarchy and importing it into Time Series Insights using the Time Series Model APIs. We recommend periodically refreshing it as updates are made in your Enterprise Asset Management system.

In the future, asset models will evolve to become digital twins, combining dynamic asset data (real-time telemetry), static data (3D models, metadata from Asset Management Systems), and graph-based relationships, allowing the digital twin to change in real-time along with the physical asset.

Azure Digital Twins is an Azure IoT service that provides the ability to:

- Create comprehensive models of physical environments,
- Create spatial intelligence graphs to model the relationships and interactions between people, places, and devices,
- Query data from a physical space rather than disparate sensors, and
- Build reusable, highly scalable, spatially aware experiences that link streaming data across the physical and digital world.

Business Process Integration

In some instances, you will want your IIoT analytics solution to perform actions based on insights from your industrial data. This can include raising alarms, sending email, sending SMS messages, or triggering a workflow in your line-of-business systems (for example, CRM, ERP, and so on). We recommend using Azure Logic Apps to

integrate your IIoT analytics solution with your line-of-business systems. Azure Logic Apps has a number of connectors to business systems and Microsoft services such as:

- Dynamics 365
- SharePoint Online
- Office 365 Outlook
- Salesforce
- SAP

For example, an error code from a pump is detected by an Azure Stream Analytics job. The job sends a message to Azure Service Bus and triggers a Logic App to run. The Logic App sends an email notification to the Plant Manager using the Office 365 Outlook connector. It then sends a message to your SAP *S/4 HANA* system using the SAP connector, which creates a Service Order in SAP.

User Management

User management involves managing user profiles and controlling what actions a user can perform in your IIoT analytics solution. For example, what asset data can a user view, or whether the user can create conditions and alerts. This is frequently referred to as role-based access control (RBAC).

We recommend implementing role-based access control using the Microsoft identity platform along with Azure Active Directory. In addition, the Azure PaaS services mentioned in this IIoT analytics solution can integrate directly with Azure Active Directory, thereby ensuring security across your solution.

Your web application and custom microservices can also integrate with the Microsoft identity platform using libraries such as Microsoft Authentication Library (or MSAL) and protocols such as OAuth 2.0 and OpenID Connect.

User management also involves operations such as:

- creating a new user,
- updating a user's profile, such as their location and phone number,
- changing a user's password, and
- disabling a user's account.

For these operations, we recommend using the Microsoft Graph.

Next steps

Data visualization is the backbone of a well-defined analytics system. Learn about the data visualization techniques that you can use with the IIoT analytics solution recommended in this series.

Data analysis in Azure Industrial IoT analytics solution

11/19/2021 • 4 minutes to read • Edit Online

This article shows you how to visualize the data collected by the Azure Industrial IoT analytics solution. You can easily look for data trends using visual elements and dashboards, and use these trends to analyze the effectiveness of your solution.

Visualization

There are many options for visualizing your industrial data. Your IIoT analytics solution may use some or all of these options, depending on the personas using your solution.

- For Process Engineers and other personas looking to perform ad-hoc analytics and trend visualizations, we recommend using Azure Time Series Insights explorer.
- For Plant Managers and other personas wanting to develop dashboards, we recommend using Power BI, and connecting Power BI with your data in Time Series Insights using the Power BI connector. Using Power BI, these users can also combine external data from your ERP, EAM, or other systems with the data in Time Series Insights.
- For advanced visualizations, such as schematic views and process graphics, we recommend a custom web application.
- For Data Scientists interested in using open source data analysis and visualization tools such as Python, Jupyter Notebooks, and *Matplotlib*, we recommend Azure Notebooks.

Data Trends

The Azure Time Series Insights explorer is a web application that provides powerful data trending and visualization capabilities that make it simple to explore and analyze billions of IIoT events simultaneously.

Time Series Insights Explorer is ideally suited to personas, such as a Process Engineer or Operations Manager, who want to explore, analyze and visualize the raw data coming from your industrial systems. The insights gained from exploring the raw data can help build Azure Stream Analytics jobs, which look for conditions in the data or perform calculations over the data.

The Azure Time Series Insights explorer allows you to seamlessly explore both warm and cold data, or your historical data, as demonstrated in the following figure.

Azure Time Series Insights explorer has a powerful yet intuitive user interface, as shown below.

Key features of Azure Time Series Insights explorer:

1. **Environment panel:** Displays all your Azure Time Series Insights environments.
2. **Navigation bar:** Lets you switch between the Analyze and Model pages.
3. **Hierarchy tree and search panel:** Lets you select and search for specific data elements to be charted.
4. **Time series well:** Shows all your currently selected data elements.
5. **Chart panel:** Displays your current working chart.
6. **Timeline:** Lets you modify your working time span.
7. **App bar:** Contains your user management options (such as current tenant), and allows you to change them and language settings.

Dashboards

For some personas, such as a Plant Manager, dashboards containing factory or plant KPIs and visualizations are more important than viewing the raw data. For such users, we recommend Power BI as the visualization solution. You can connect Power BI with your data stored in Time Series Insights, providing you with powerful reporting and dashboard capabilities over your industrial data, and allowing you to share insights and results across your organization.

By connecting your data to Power BI, you can:

- Perform correlations with other data sources supported by Power BI and access a host of different data visualization options.
- Create Power BI dashboards and reports using your Time Series Insight data and share them with your organization.
- Unlock data interoperability scenarios in a simple, easy-to-use manner, and get to insights faster than ever.
- Modify Time Series Insights data within Power BI using the powerful Advanced Editor.

Schematic Views

For advanced visualizations, such as schematic views or process graphics, you may require a custom web application. A custom web application also allows you to provide a *single pane of glass* user experience and other advanced capabilities including:

- a simplified and integrated authoring experience for Azure Stream Analytics jobs and Logic Apps,
- displaying real-time data using process or custom visuals,
- displaying KPIs and external data with embedded Power BI dashboards,
- displaying visual alerts using SignalR, and
- allowing administrators to add/remove users from the solution.

We recommend building a Single Page Application (SPA) using:

- JavaScript, HTML5, and CSS3
- Time Series Insights JavaScript SDK for displaying process or custom visuals with data from Time Series Insights
- MSAL.js to sign in users and acquire tokens to use with the Microsoft Graph
- Azure App Services Web Apps to host the web application
- Power BI to embed Power BI dashboards directly in the web app
- Azure Maps to render map visualizations
- Microsoft Graph SDK for JavaScript to integrate with Microsoft 365

Notebooks

One of the advantages of moving operational data to the cloud is to take advantage of modern big-data tool sets. One of the most common tools used by Data Scientists for ad-hoc analysis of big data are Jupyter Notebooks. Jupyter (formerly IPython) is an open-source project that lets you easily combine Markdown text,

executable code, persistent data, graphics, and visualizations onto a single, sharable canvas - *the notebook*. Production Engineers should also consider learning Jupyter Notebooks technology to assist in analysis of plant events, finding correlations, and so on. Jupyter Notebooks provide support for Python 2/3, R, and F# programming languages and can connect to your Time Series Insights data stored Azure storage.

Next steps

Now that you have learned the architecture of an Azure IIoT analytics solution, read the architectural considerations that improve the resiliency and efficiency of this architecture.

Architectural considerations in an IIoT analytics solution

11/19/2021 • 3 minutes to read • Edit Online

The Microsoft Azure Well-Architected Framework describes some key tenets of a good architectural design. Keeping in line with these tenets, this article describes the considerations in the reference Azure Industrial IoT analytics solution that improve its performance and resiliency.

Performance considerations

Azure PaaS services

All Azure PaaS services have an ability to scale up and/or out. Some services will do this automatically (for example, IoT Hub, Azure Functions in a Consumption Plan) while others can be scaled manually.

As you test your IIoT analytics solution, we recommend that you do the following:

- Understand how each service scales (and what the units of scale are).
- Collect performance metrics and establish baselines.
- Set up alerts when performance metrics exceed baselines.

All Azure PaaS services have a metrics blade that allows you to view service metrics, and configure conditions and alerts, which are collected and displayed in Azure Monitor. We recommend enabling these features to ensure your solution performs as expected.

IoT Edge

Azure IoT Edge gateway performance is impacted by the following:

- The number of edge modules running and their performance requirements
- The number of messages processed by modules and EdgeHub
- Edge modules requiring GPU processing
- Offline buffering of messages
- The gateway hardware
- The gateway operating system

We recommend real world testing and/or testing with simulated telemetry to understand the field gateway hardware requirements for Azure IoT Edge. Conduct your initial testing using virtual machine where CPU, RAM, disk can be easily adjusted. Once approximate hardware requirements are known, get your field gateway hardware and conduct your testing again using actual hardware.

You should also test to ensure the following:

- No messages are being lost between source (for example, historian) and destination (for example, Time Series Insights).
- Acceptable message latency exists between the source and the destination.
- Source timestamps are preserved.
- Data accuracy is maintained, especially when performing data transformations.

Availability considerations

IoT Edge

A single Azure IoT Edge field gateway can be a single point of failure between your SCADA, MES, or historian and Azure IoT Hub. A failure can cause gaps in data in your IIoT analytics solution. To prevent this, IoT Edge can integrate with your on-premise Kubernetes environment, using it as a resilient, highly available infrastructure layer. For more information, see How to install IoT Edge on Kubernetes (Preview).

Network considerations

IoT Edge and firewalls

To maintain compliance with standards such as ISA 95 and ISA 99, industrial equipment is often installed in a closed Process Control Network (PCN), behind firewalls, with no direct access to the Internet (see Purdue networking model).

There are three options to connect to equipment installed in a PCN:

1. Connect to a higher-level system, such as a historian, located outside of the PCN.
2. Deploy an Azure IoT Edge device or virtual machine in a DMZ between the PCN and the internet.
   a. The firewall between the DMZ and the PCN will need to allow inbound connections from the DMZ to the appropriate system or device in the PCN.
   b. There may be no internal DNS setup to resolve PCN names to IP addresses.
3. Deploy an Azure IoT Edge device or virtual machine in the PCN and configure IoT Edge to communicate with the Internet through a Proxy server.
   a. Additional IoT Edge setup and configuration are required. See Configure an IoT Edge device to communicate through a proxy server.
   b. The Proxy server may introduce a single point of failure and/or a performance bottleneck.
   c. There may be no DNS setup in the PCN to resolve external names to IP addresses.

Azure IoT Edge will also require:

- access to container registries, such as Docker Hub or Azure Container Registry, to download modules over HTTPS,
- access to DNS to resolve external FQDNs, and
- ability to communicate with Azure IoT Hub using MQTT, MQTT over WebSockets, AMQP, or AMQP over WebSockets.

For additional security, industrial firewalls can be configured to only allow traffic between IoT Edge and IoT Hub using Service Tags. IP address prefixes of IoT Hub public endpoints are published periodically under the *AzureIoTHub* service tag. Firewall administrators can programmatically retrieve the current list of service tags, together with IP address range detail, and update their firewall configuration.

Next steps

- For a more detailed discussion of the recommended architecture and implementation choices, download and read the Microsoft Azure IoT Reference Architecture pdf.
- Azure Industrial IoT components, tutorials, and source code.
- For detailed documentation of the various Azure IoT services, see Azure IoT Fundamentals.

Azure IoT client SDK support for third-party token servers

11/19/2021 • 9 minutes to read • Edit Online

The article Control access to IoT Hub illustrates how a third-party token service can be integrated with IoT Hub. This article outlines the support for shared access signature (SAS) token authentication in each of the Azure IoT client SDKs. It also outlines both what needs to be implemented in a device application using the corresponding SDK for each language, and how to use device-scoped or module-scoped tokens for the shared access policies of DeviceConnect or ModuleConnect.

Context and problem

The current Azure IoT Hub security documentation discusses the third-party token-server pattern for SAS authentication with IoT Hub by IoT devices using the Azure IoT client SDKs. However, incorrect assumptions made by a customer during a recent enterprise engagement suggests that without further clarification, you can develop a misleading impression about the level of support implemented by default in the Azure IoT client SDKs.

This article discusses the learning from that engagement and clarifies what needs to be done in each SDK for devices to achieve third-party token-server authentication. This article should also prevent you from making similar incorrect assumptions about support for the third-party token-server pattern in the Azure IoT client SDK.

Solution

The Azure IoT client SDKs provide varying levels of support for SAS token authentication, each requiring some custom code to complete the authentication and token management functionality.

The token evaluation frequency depends on the chosen transport protocol—MQTT, AMQP, or HTTPS. The variation depends on the capability of the protocol to support proactive renewal of tokens and session time-outs. Only AMQP implements proactive renewal support. This means the other transports will close the connection on SAS token authentication failure, and then need to perform a new connection operation. This is a potentially expensive connectivity operation for the client.

If SAS authentication fails, an error is raised by the transport implementation that can be handled within the device application by a "Connection Status Changed" event handler. Failure to implement such a handler will typically see the device application halt due to the error. With the correct implementation of the event handler and token renewal functionality, the transports can re-attempt the connection.

The following figure illustrates the third-party token-server pattern:

The following figure illustrates implementation support in the Azure IoT client SDK with Mobile Net Operator integration:

Sample implementations are included in the Azure Samples repository on GitHub.

Issues and considerations

Consider the following points when deciding whether to implement this pattern:

- The Azure IoT Hub Device Provisioning Service (Azure DPS) client SDKs do not support SAS token authentication. The Azure DPS REST API *does* support SAS token authentication. Therefore, to use the Azure DPS with a third-party token service for SAS authentication, a device application must implement the device DPS process using the Azure DPS REST API.
- This consists of making an initial registration request operation, and then polling the operational status

API until the DPS process succeeds or fails. On success, the device provisioning details can be obtained by requesting them from the Azure DPS REST API Runtime Registration.

References:

- Initial registration request API
- Poll operation API
- Request provisioned device details API

When to use this pattern

You should user this pattern whenever you want to authenticate to Azure IoT Hub from IoT devices using the various Azure IoT Client SDKs. Instead of using the client SDKs for SAS token authentication, use the Azure DPS REST API to ensure implementation of proactive renewal support for all transport mechanisms.

Examples

The following sections offer examples that you can use for different programming languages, such as Embedded C, .NET, Java, and Python.

Azure IoT Hub device SDK for C and Azure IoT Hub device SDK for Embedded C

The following approach can be utilized in device applications built using the Azure IoT C SDK or the Azure IoT Embedded C SDK. Neither SDK provides SAS token lifetime management, therefore you'll need to implement a SAS token lifetime manager capability.

SAS tokens can be used via the *IOTHUB_CLIENT_CONFIG* structure by setting the *deviceSasToken* member to the token and making the *deviceKey* null. Other unused values, such as *protocolGatewayHostName*, must also be set to null.

```
CONFIG = (IOTHUB\_CLIENT\_CONFIG\*)MALLOC(SIZEOF (IOTHUB\_CLIENT\_CONFIG));

CONFIG-\>PROTOCOL = PROTOCOL;

CONFIG-\>DEVICEID = DEVICEID;

CONFIG-\>IOTHUBNAME = IOTHUBNAME;

CONFIG-\>IOTHUBSUFFIX = IOTHUBSUFFIX;

CONFIG-\>DEVICEKEY = 0;

CONFIG-\>DEVICESASTOKEN = TOKEN;

CONFIG-\>PROTOCOLGATEWAYHOSTNAME = 0;
`The created IOTHUB\_CLIENT\_CONFIG can then be provided to the IoTHubDeviceClient\_Create function to
establish a DeviceClient instance.`
IF ((IOTHUBCLIENTHANDLE = IOTHUBDEVICECLIENT\_CREATE(CONFIG)) == NULL)

{

(VOID)PRINTF("ERROR: IOTHUBCLIENTHANDLE IS NULL!\\R\\N");

}
`To capture SAS token authentication failures, a handler needs to be implemented for the
IoTHubDeviceClient\_SetConnectionStatusCallback.`
(VOID)IOTHUBDEVICECLIENT\_SETCONNECTIONSTATUSCALLBACK(IOTHUBCLIENTHANDLE,
CONNECTION\_STATUS\_CALLBACK, NULL);
```

The connection_status_callback can catch the IOTHUB_CLIENT_CONNECTION_STATUS_REASON of

IOTHUB_CLIENT_CONNECTION_EXPIRED_SAS_TOKEN to trigger a renewal of the SAS token via the third-party token service. This is required for all transports to capture connection issues but is specifically required by transports which do not support proactive SAS token renewal. Proactive SAS token lifetime management can be implemented as a function repeatedly run during the device applications "operational" loop. Ensuring the lifetime of the token is frequently evaluated, and token renewal can be proactively executed when required.

SAS token authentication implementation summary for C SDKs:

1. Implement a ConnectionStatusCallback handler to capture IOTHUB_CLIENT_CONNECTION_EXPIRED_SAS_TOKEN event and trigger token renewal.
2. Use an IOTHUB_CLIENT_CONFIG to provide the device SAS token to IoTHubDeviceClient_Create.
3. Implement proactive SAS token lifetime management as part of the device application's operation loop.

References:

- IoTHubDeviceClient_Create() reference
- IoTHubDeviceClient_SetConnectionStatusCallback() reference

Azure IoT Hub device SDK for .Net

The Azure IoT client SDK for .Net implements support for SAS token lifetime management through the abstract DeviceAuthenticationWithTokenRefresh class. A concrete implementation of this class, adding token renewal functionality, can be provided as the authentication method to a DeviceClient.Create method. The transport implementations will automatically renew the token via the authentication method as required. A ConnectionStatusChangesHandler is required to capture connection changes and prevent exceptions being raised by the transports.

Example implementation based on the DeviceAuthenticationWithTokenRefreash class

```
INTERNAL CLASS STSDEVICEAUTHENTICATIONWITHTOKENREFRESH :
DEVICEAUTHENTICATIONWITHTOKENREFRESH

{

PRIVATE READONLY STRING \_STSCONNECTURL =
"HTTP://LOCALHOST:8080/STS/AZURE/TOKEN/OPERATIONS?SR={0}/DEVICES/{1}";

PRIVATE CONST INT DEFAULTTIMETOLIVESECONDS = 1 \* 60 \* 60;

PRIVATE CONST INT DEFAULTBUFFERPERCENTAGE = 15;

PUBLIC STSDEVICEAUTHENTICATIONWITHTOKENREFRESH(STRING DEVICEID, INT
SUGGESTEDTIMETOLIVESECONDS, INT TIMEBUFFERPERCENTAGE) : BASE(DEVICEID,
SUGGESTEDTIMETOLIVESECONDS, TIMEBUFFERPERCENTAGE)

{

IF (STRING.ISNULLORWHITESPACE(DEVICEID))

{

THROW NEW ARGUMENTNULLEXCEPTION(NAMEOF(DEVICEID));

}

}

PROTECTED OVERRIDE TASK\<STRING\> SAFECREATENEWTOKEN(STRING IOTHUB, INT
SUGGESTEDTIMETOLIVE)

{
```

```
STRING RESULT;

STRING URL = STRING.FORMAT(\_STSCONNECTURL, IOTHUB, DEVICEID);

HTTPCLIENTHANDLER HANDLER = NEW HTTPCLIENTHANDLER();

HTTPCLIENT = NEW HTTPCLIENT(HANDLER);

TRY

{

VAR APIRESPONSE = HTTPCLIENT.GETASYNC(URL).RESULT;

IF (APIRESPONSE.ISSUCCESSSTATUSCODE)

{

RESULT = APIRESPONSE.CONTENT.READASSTRINGASYNC().RESULT;

}

ELSE

{

THROW NEW HTTPREQUESTEXCEPTION();

}

}

CATCH (HTTPREQUESTEXCEPTION)

{

RESULT = NULL;

}

RETURN TASK.FROMRESULT(RESULT);

}

}
```

SAS token authentication implementation summary for Azure IoT Hub device SDK for .Net:

1. Implement a concrete class based on the DeviceAuthenticationWithTokenRefresh abstract class, which implements token renewal functionality.
2. Implement a ConnectionStatusChangesHandler to capture transport connection status and avoid exceptions raised by transport implementation.

References:

- DeviceAuthenticationWithTokenRefresh Class
- DeviceClient.Create Method

Azure IoT Hub device SDK for Java

The Azure IoT Client SDK for Java implements support for SAS token lifetime management through the SasTokenProvider Interface. A class that implements this interface with SAS token renewal functionality can be used as the SecurityProvider in a DeviceClient constructor. The transport implementations will automatically

renew the token via the security provider as required. A ConnectionStatusChangeCallback needs to be registered to capture connection changes and prevent exceptions being raised by the transports.

Example implementation of the security provider implementing the SasTokenProvider interface:

```
IMPORT JAVA.IO.IOEXCEPTION;

IMPORT JAVA.NET.URI;

IMPORT JAVA.NET.HTTP.HTTPCLIENT;

IMPORT JAVA.NET.HTTP.HTTPREQUEST;

IMPORT JAVA.NET.HTTP.HTTPRESPONSE;

IMPORT JAVA.TIME.DURATION;

PUBLIC CLASS STSSECURITYPROVIDER IMPLEMENTS SASTOKENPROVIDER {

PRIVATE FINAL STRING HOSTNAME;

PRIVATE FINAL STRING DEVICEID;

PRIVATE INT RENEWALBUFFERSECONDS;

PRIVATE LONG EXPIRYTIMESECONDS;

PRIVATE CHAR[] SASTOKEN;

PUBLIC STSSECURITYPROVIDER(STRING HOSTNAME, STRING DEVICEID)

{

THIS.HOSTNAME = HOSTNAME;

THIS.DEVICEID = DEVICEID;

THIS.RENEWALBUFFERSECONDS = 120;

THIS.EXPIRYTIMESECONDS = (SYSTEM.CURRENTTIMEMILLIS() / 1000);

}

\@OVERRIDE

PUBLIC CHAR[] GETSASTOKEN() {

LONG CURRENTTIMESECONDS = (SYSTEM.CURRENTTIMEMILLIS() / 1000);

TRY {

IF (THIS.SASTOKEN == NULL \|\| THIS.EXPIRYTIMESECONDS +
THIS.RENEWALBUFFERSECONDS \>= CURRENTTIMESECONDS) {

THIS.SASTOKEN = STSGETTOKEN();

ASSERT THIS.SASTOKEN != NULL;

STRING T = STRING.COPYVALUEOF(THIS.SASTOKEN);

STRING[] BITS = T.SPLIT("SE=");

LONG L = LONG.PARSELONG(BITS[1]);

THIS.EXPIRYTIMESECONDS = L; // THE SE= NUMBER

THIS.RENEWALBUFFERSECONDS = (INT) (L \* 0.15); // RENEW WITHIN 15% OF EXPIRY
```

```
}

} CATCH (INTERRUPTEDEXCEPTION \| IOEXCEPTION E) {

E.PRINTSTACKTRACE();

}

RETURN THIS.SASTOKEN;

}

PRIVATE CHAR[] STSGETTOKEN() THROWS IOEXCEPTION, INTERRUPTEDEXCEPTION {

STRING STSURL =
STRING.FORMAT("HTTP://LOCALHOST:8080/STS/AZURE/TOKEN/OPERATIONS?SR=%S/DEVICES/%S",
THIS.HOSTNAME, THIS.DEVICEID);

HTTPREQUEST REQUEST = HTTPREQUEST.NEWBUILDER()

.URI(URI.CREATE(STSURL))

.TIMEOUT(DURATION.OFMINUTES(2))

.HEADER("CONTENT-TYPE", "APPLICATION/JSON")

.BUILD();

HTTPCLIENT CLIENT = HTTPCLIENT.NEWBUILDER()

.VERSION(HTTPCLIENT.VERSION.HTTP\_1\_1)

.CONNECTTIMEOUT(DURATION.OFSECONDS(20))

.BUILD();

HTTPRESPONSE\<STRING\> RESPONSE = CLIENT.SEND(REQUEST,
HTTPRESPONSE.BODYHANDLERS.OFSTRING());

IF(RESPONSE.STATUSCODE() \<200 \|\| RESPONSE.STATUSCODE()\>=300) {

RETURN NULL;

}

IF(RESPONSE.BODY().ISEMPTY()) {

RETURN NULL;

}

RETURN RESPONSE.BODY().TOCHARARRAY();

}

}
```

SAS token authentication implementation summary for Azure IoT Hub device SDK for Java:

1. Implement the SasTokenProvider interface on a class and include token renewal functionality.
2. Implement a ConnectionStatusChangeCallback handler to capture transport connection status changes and avoid exceptions raised by transport implementation.

References:

- SasTokenProvider Interface
- DeviceClient.registerConnectionStateCallback(IotHubConnectionStateCallback callback, Object callbackContext) Method
- Custom SAS token provider sample - SDK Sample

Azure IoT Hub device SDK for Python

The Azure IoT Hub device SDK for Python implements SAS token support through methods on the IoTHubDeviceClient object. These methods enable the creation of a device client using a token, and the ability to supply an updated token once the device client has been created. They do not implement token lifetime management, but this can be implemented easily as an asynchronous operation.

A Python 3.7 example implementation showing just the outline of functionality:

```
ASYNC DEF MAIN():

\# GET A SASTOKEN YOU GENERATED

SASTOKEN = GET\_NEW\_SASTOKEN()

\# THE CLIENT OBJECT IS USED TO INTERACT WITH YOUR AZURE IOT HUB.

DEVICE\_CLIENT = IOTHUBDEVICECLIENT.CREATE\_FROM\_SASTOKEN(SASTOKEN)

\# CONNECT THE CLIENT.

AWAIT DEVICE\_CLIENT.CONNECT()

\# DEFINE BEHAVIOR FOR PROVIDING NEW SASTOKENS TO PREVENT EXPIRY

ASYNC DEF SASTOKEN\_KEEPALIVE():

WHILE TRUE:

AWAIT ASYNCIO.SLEEP(NEW\_TOKEN\_INTERVAL)

SASTOKEN = GET\_NEW\_SASTOKEN()

AWAIT DEVICE\_CLIENT.UPDATE\_SASTOKEN(SASTOKEN)

\# ALSO RUN THE SASTOKEN KEEPALIVE IN THE EVENT LOOP

KEEPALIVE\_TASK = ASYNCIO.CREATE\_TASK(SASTOKEN\_KEEPALIVE())

\# CANCEL THE SASTOKEN UPDATE TASK

KEEPALIVE\_TASK.CANCEL()

\# FINALLY, SHUT DOWN THE CLIENT

AWAIT DEVICE\_CLIENT.SHUTDOWN()

IF \_\_NAME\_\_ == "\_\_MAIN\_\_":

ASYNCIO.RUN(MAIN())
```

Summary of Azure IoT Hub device SDK for Python SAS token authentication:

1. Create SAS token generation function.
2. Create a device client using IoTHubDeviceClient.create_from_sastoken.
3. Manage token lifetime as a separate activity, supplying the device client with a renewed token when

required by the IoTHubDeviceClient.update_sastoken method.

References:

- IoTHubDeviceClient

Azure IoT Hub device SDK for Node.JS/JavaScript

The Azure IoT for Node.JS/JavaScript implements a SharedAccessSignatureAuthenticationProvider that will serve an SAS token to the device client and transports to authenticate with IoT Hub. It does not implement any token renewal functionality. The device application must manage token lifetime, renewing the token as required.

Use the device client methods fromSharedAccessSignature and updateSharedAccessSignature to initiate a connection with IoT Hub and supply a renewed token to the SharedAccessSignatuteAuthenticationProvider, which will cause the authentication provider to emit a newTokenAvailable event to the transports.

A basic SAS token sample is provided in the simple_sample_device_with_sas.js example.

Summary of Azure IoT Hub device SDK for Node.JS/JavaScript

1. Implement SAS token lifetime management and renewal.
2. Use device client fromSharedAccessSignature to construct a device client instance.
3. Use device client updateSharedAccessSignature to supply a renewed token.

References:

- Client class

Next steps

- Control access to IoT Hub using Shared Access Signatures and security tokens
- Communicate with your IoT hub using the MQTT protocol
- Communicate with your IoT hub by using the AMQP Protocol
- Azure IoT Hub SDKs

Related resources

- Azure IoT Hub developer guide
- Choose an Internet of Things (IoT) solution in Azure
- Getting started with Azure IoT solutions

Compare Internet of Things (IoT) solution approaches (PaaS vs. aPaaS)

11/19/2021 • 7 minutes to read • Edit Online

IoT solutions require a combination of technologies to effectively connect devices, events, and actions to cloud applications. In Azure, we have a single set of guidance for building and connecting devices to the cloud. However, there are many options for building and deploying your IoT cloud solutions. Which technologies and services you'll use depends on your scenario's development, deployment, and management needs.

Starting with Azure IoT Central (aPaaS)

Using an application platform as a service (aPaaS) streamlines many of the complex decisions you'll face when building an IoT solution. Azure IoT Central is Microsoft's aPaaS offering designed to simplify and accelerate IoT solution assembly and operation by assembling platform as a service (PaaS) components into an extensible and fully managed app development platform hosted by Microsoft. This takes a lot of the guesswork and complexity out of building reliable, scalable, and secure IoT applications.

An out-of-the box web UX and API surface area make it simple to monitor device conditions, create rules, and manage millions of devices and their data remotely throughout their life cycles. Furthermore, it enables you to act on device insights by extending IoT intelligence into line-of-business applications. Azure IoT Central also offers built-in disaster recovery, multitenancy, global availability, and a predictable cost structure.

Building with Azure PaaS services

In certain scenarios you may need a higher degree of control and customization than Azure IoT Central provides. In these cases, Azure also offers individual platform as a service (PaaS) cloud services that you can use to build a custom IoT solution. For example, you might build a solution using a combination of these PaaS services:

- Azure IoT Device Provisioning Service and Azure IoT Hub for device provisioning, device connectivity, and management
- Azure Time Series Insights for storing and analyzing warm and cold path time series data from IoT devices

- Azure Stream Analytics for analyzing hot path data from IoT devices
- Azure IoT Edge for running AI, third-party services, or your own business logic on IoT Edge devices

Comparing approaches

Choosing to build with Azure IoT Central gives you the opportunity to focus time and money on transforming your business and designing innovative offerings, rather than maintaining and updating a complex and continually evolving IoT infrastructure. However, if your solution requires features or services that Azure IoT Central does not currently support, you may need to develop a PaaS solution using Azure IoT Hub as a core element.

You can use the table and links below to help decide if you can use a managed solution based on Azure IoT Central, or if you should consider building a PaaS solution using Azure IoT Hub.

	AZURE IOT CENTRAL	AZURE IOT HUB

	AZURE IOT CENTRAL	AZURE IOT HUB
Type of Service	Fully managed aPaaS solution. It simplifies device connectivity and management at scale so that you can focus time and resources on using IoT for business transformation. This simplicity comes with a tradeoff: an aPaaS-based solution is less customizable than a PaaS-based solution.	Managed PaaS back-end solution that acts as a central message hub between your IoT application and the devices it manages. You can build more functionality using additional Azure PaaS services. This approach provides great flexibility but requires more development and management effort to build and operate your solution.
Application Template	Application templates in Azure IoT Central help solution builders kick-start IoT solution development. You can get started with a generic application template, or use a prebuilt industry-focused application template for retail, energy, government, or healthcare.	Not supported. You'll design and build your own solution using Azure IoT Hub and other PaaS services.
Device Management	Provides seamless device integration and device management capability. Device Provisioning Service capabilities (DPS) are built in.	No built-in experience. You'll design and build your own solutions using Azure IoT Hub primitives, such as device twin and direct methods. DPS must be enabled separately.
Scalability	Supports auto-scaling.	There is no built-in mechanism for automatically scaling an IoT Hub. You'll need to deploy other solutions to enable auto-scaling. See: Auto-scale your Azure IoT Hub
Message Retention	Retains data on a rolling, 30-day basis. You can continuously export data using the export feature.	Allows data retention in the built-in Event Hubs for a maximum of 7 days.
Visualizations	Yes, IoT Central has a UX that makes it simple to visualize device data, perform analytics queries, and create custom dashboards.	No.
OPC UA Protocol	Not yet.	OPC Publisher is a Microsoft-supported open-source product that bridges the gap between industrial assets and Azure hosted resources. It connects to OPC UA–enabled assets or industrial connectivity software and publishes telemetry data to Azure IoT Hub in various formats, including IEC62541 OPC UA PubSub standard format. See: Azure/iot-edge-opc-publisher: Microsoft OPC Publisher

	AZURE IOT CENTRAL	AZURE IOT HUB
Pricing	The first two active devices within an IoT Central application are free, if their message volume does not exceed 800 (Standard Tier 0 plan), 10,000 (Standard Tier 1 plan), or 60,000 (Standard Tier 2 plan) per month. Volumes exceeding those thresholds will incur overage charges. Beyond that, device pricing is prorated monthly. For each hour during the billing period, the highest number of active devices is counted and billed. See: Azure IoT Central pricing	See: Azure IoT Hub pricing
Analytics, Insights, and Actions	Integrated analytics experience targeted at exploration of device data in the context of device management.	You'll use separate Azure PaaS services to incorporate analytics, insights, and actions, like Azure Steam Analytics, Time Series Insight, Azure Data Explorer, and Azure Synapse.
Big Data Management	Data Management can be managed from Azure IoT Central itself.	You'll need to add and manage big data Azure PaaS services as part of your solution.
High Availability and Disaster Recovery	High availability and disaster recovery capabilities are built in to Azure IoT Central and managed for you automatically. See: Best practices for device development in Azure IoT Central	Can be configured to support multiple high availability and disaster recovery scenarios. See: Azure IoT Hub high availability and disaster recovery
SLA	Azure IoT Central guarantees you 99.9% connectivity. See: SLA for Azure IoT Central	The Azure IoT Hub standard and basic tiers guarantee 99.9% uptime. No SLA is provided for the Free Tier of Azure IoT Hub. See: SLA for Azure IoT Hub
Device Template	Supports centrally defining and managing device templates that help structure the characteristics and behaviors of device types for use in supported device management tasks and visualizations.	Requires users to create their own repository to define and manage device message templates.
Data Export	Provides data export to Azure blob storage, event hubs, service bus, webhook, and Azure Data Explorer. Additional capabilities include filtering, enriching, and transforming messages on egress.	Provides a built-in event hub endpoint and can also make use of message routing to export data to other storage locations.

	AZURE IOT CENTRAL	AZURE IOT HUB
Multi-tenancy	IoT Central Organizations enabled in-app multi-tenancy where you to define a hierarchy to manage which users can see which devices in your IoT Central application.	Not supported. Tenancy can be achieved by using separate hubs per customer and/or access control can be built into the data layer of solutions.
Rules and Actions	Provides a built-in rules and actions processing capability with email notification, Azure Monitor group, Power Automate, and Webhook actions. See: What is Azure IoT Central?	Data coming from IoT Hub can be sent to Azure Stream Analytics, Azure Time Series Insights, or Azure Event Grid. From those services you can connect to Azure Logic apps or other custom applications to handle rules and actions processing. See: IoT remote monitoring and notifications with Azure Logic Apps
SigFox/LoRaWAN Protocol	Uses IoT Central Device Bridge. See: Azure IoT Central Device Bridge	Requires you to write a custom Module on Azure IoT Edge and integrate it with Azure IoT Hub.

Next steps

Continue learning about IoT Hub and IoT Central:

- What is Azure IoT Central?
- What is Azure IoT Hub?

Related resources

Additional IoT topics:

- Overview of device management with Azure IoT Hub
- Azure IoT Hub high availability and disaster recovery
- Understand and use Azure IoT Hub SDKs
- IoT remote monitoring and notifications with Azure Logic Apps

IoT architecture guides:

- IoT solutions conceptual overview
- Vision with Azure IoT Edge
- Azure Industrial IoT Analytics Guidance
- Azure IoT reference architecture
- IoT and data analytics

Example architectures using Azure IoT Central:

- Retail - Buy online, pickup in store (BOPIS)
- Environment monitoring and supply chain optimization with IoT
- Blockchain workflow application

Example architectures using Azure IoT Hub:

- Azure IoT reference architecture
- IoT and data analytics
- IoT using Cosmos DB
- Predictive maintenance with the intelligent IoT Edge
- Predictive Maintenance for Industrial IoT
- Project 15 Open Platform
- IoT connected light, power, and internet for emerging markets
- Condition Monitoring for Industrial IoT

Move an IoT solution from test to production

11/19/2021 • 5 minutes to read • Edit Online

This article includes a list of items you should consider when moving an IoT solution to a production environment.

Use deployment stamps

Stamps are discrete units of core solution components that support a defined number of devices. Each copy is called a *stamp*. or *scale unit*. For example, a stamp might consist of a set device population, an IoT Hub, an Event Hub or other routing endpoint, and a processing component. Each stamp supports a defined device population. You choose the maximum number of devices the stamp can hold. As the device population grows, you add stamp instances rather than independently scaling up different parts of the solution.

If instead of adding stamps, you move a single instance of your IoT solution to production, you might encounter the following limitations:

- **Scale limits:** Your single instance can encounter scaling limits. For example, your solution might be using services that have limits on the number of inbound connections, host names, TCP sockets, or other resources.
- **Non-linear scaling or cost:** Your solution components may not scale linearly with the number of requests made or the amount of data ingested. Instead, for some components, there might be a decrease in performance or increase in cost once a threshold has been met. Scaling up with more capacity might not be as good a strategy as scaling out by adding stamps.
- **Separation of Customers:** You may need to keep certain customers' data isolated from other customers' data. Similarly, you may have some customers that require more system resources to service than others, and consider grouping them on different stamps.
- **Single and multi-tenant instances:** You might have some large customers who need their own independent instances of your solution. You might also have a pool of smaller customers who can share a multi-tenant deployment.
- **Complex deployment requirements:** You might need to deploy updates to your service in a controlled manner and deploy to different stamps at different times.
- **Update frequency:** You might have some customers who are tolerant of having frequent updates to your system, while others might be risk-averse and want infrequent updates to your service.
- **Geographical or geopolitical restrictions:** To reduce latency or comply with data sovereignty requirements, you can deploy some of your customers into specific regions.

To avoid the preceding issues, consider grouping your service into multiple stamps. Stamps operate independently of each other and can be deployed and updated independently. A single geographical region may contain a single stamp, or may contain multiple stamps to allow for horizontal scale-out within the region. Each stamp contains a subset of your customers.

Scale IoT solutions with stamps

Use back-off when a transient fault occurs

All applications that communicate with remote services and resources must be sensitive to transient faults. This

is especially the case for applications that run in the cloud, where the nature of the environment and connectivity over the internet means these types of faults are likely to be encountered more often. Transient faults include:

- Momentary loss of network connectivity to components and services
- Temporary unavailability of a service
- Timeouts that arise when a service is busy
- Collisions caused when devices transmit simultaneously

These faults are often self-correcting, and if the action is repeated after a suitable delay it is likely to succeed. Determining the appropriate intervals between retries is, however, difficult. Typical strategies use the following types of retry intervals:

- **Exponential back-off**. The application waits a short time before the first retry, and then exponentially increasing times between each subsequent retry. For example, it may retry the operation after 3 seconds, 12 seconds, 30 seconds, and so on.
- **Regular intervals**. The application waits for the same period of time between each attempt. For example, it may retry the operation every 3 seconds.
- **Immediate retry**. Sometimes a transient fault is brief, perhaps due to an event such as a network packet collision or a spike in a hardware component. In this case, retrying the operation immediately is appropriate because it may succeed if the fault has cleared in the time it takes the application to assemble and send the next request. However, there should never be more than one immediate retry attempt, and you should switch to alternative strategies, such as exponential back-off or fallback actions, if the immediate retry fails.
- **Randomization**. Any of the preceding retry strategies may include a randomization element to prevent multiple instances of the client sending subsequent retry attempts at the same time.

Avoid also the following anti-patterns:

- Implementations should not include duplicated layers of retry code.
- Never implement an endless retry mechanism.
- Never perform an immediate retry more than once.
- Avoid using a regular retry interval.
- Prevent multiple instances of the same client, or multiple instances of different clients, from sending retries at the same times.

Transient fault handling

Use zero-touch provisioning

Provisioning is the act of enrolling a device into Azure IoT Hub. Provisioning makes IoT Hub aware of the device and the attestation mechanism the device uses. You can use the Azure IoT Hub Device Provisioning Service (DPS) or provision directly via IoT Hub Registry Manager APIs. Using DPS confers the benefit of late binding, which allows removing and reprovisioning field devices to IoT Hub without changing the device software.

The following example shows how to implement a test-to-production environment transition workflow by using DPS.

1. The solution developer links the Test and Production IoT clouds to the provisioning service.
2. The device implements the DPS protocol to find the IoT Hub if it's no longer provisioned. The device is initially provisioned to the Test environment.
3. Since the device is registered with the Test environment, it connects there and testing occurs.
4. The developer re-provisions the device to the Production environment and removes it from the Test hub. The Test hub rejects the device the next time it reconnects.
5. The device connects and re-negotiates the provisioning flow. DPS now directs the device to the Production environment, and the device connects and authenticates there.

Overview of Azure IoT Hub Device Provisioning Service

Next steps

- Getting started with Azure IoT solutions
- IoT solutions conceptual overview
- Azure Industrial IoT analytics guidance
- Choose an Internet of Things (IoT) solution in Azure
- Azure IoT reference architecture

Azure mainframe and midrange architecture concepts and patterns

11/19/2021 • 12 minutes to read • Edit Online

Mainframe and midrange hardware is composed of a family of systems from various vendors (all with a history and goal of high performance, high throughput, and sometimes high availability). These systems were often *scale-up* and monolithic, meaning they were a single, large frame with multiple processing units, shared memory, and shared storage.

On the application side, programs were often written in one of two flavors: either transactional or batch. In both cases, there were a variety of programming languages that were used, including COBOL, PL/I, Natural, Fortran, REXX, and so on. Despite the age and complexity of these systems, there are many migration pathways to Azure.

On the data side, data is usually stored in files and in databases. Mainframe and midrange databases commonly come in a variety of possible structures, such as relational, hierarchical, and network, among others. There are different types of file organizational systems, where some of them can be indexed and can act as a key-value stores. Further, data encoding in mainframes can be different from the encoding usually handled in non-mainframe systems. Therefore, data migrations should be handled with upfront planning. There are many options for migrating to the Azure data platform.

Migrating legacy systems to Azure

In many cases, mainframe, midrange, and other server-based workloads can be replicated in Azure with little to no loss of functionality. Sometimes users do not notice changes in their underlying systems. In other situations, there are options for refactoring and re-engineering the legacy solution into an architecture that is in alignment with the cloud. This is done while still maintaining the same or very similar functionality. The architectures in this content set (plus the additional white papers and other resources provided below) will help guide you through this process.

Mainframe and midrange concepts

In our mainframe architectures, we use the following terms.

Mainframes

Mainframes were designed as scale-up servers to run high-volume online transactions and batch processing in the late 1950s. As such, mainframes have software for online transaction forms (sometimes called green screens) and high-performance I/O systems, for processing the batch runs. Mainframes have a reputation for high reliability and availability, in addition to their ability to run online and batch jobs.

Mainframe storage

Part of demystifying mainframes involves decoding various overlapping terms. For example, central storage, real memory, real storage, and main storage generally all refer to storage that is attached directly to the

mainframe processor. Mainframe hardware includes processors and many other devices, such as direct-access storage devices (DASDs), magnetic tape drives, and several types of user consoles. Tapes and DASDs are used for system functions and by user programs.

Types of physical storage:

- **Central storage**. Located directly on the mainframe processor, it's also known as *processor storage* or *real storage*.
- **Auxiliary storage**. Located separately from the mainframe, it includes storage on DASDs, which is also known as *paging storage*.

MIPS

The measurement of millions of instructions per second (MIPS) provides a constant value of the number of cycles per second, for a given machine. MIPS are used to measure the overall compute power of a mainframe. Mainframe vendors charge customers, based on MIPS usage. Customers can increase mainframe capacity to meet specific requirements. IBM maintains a processor capacity index, which shows the relative capacity across different mainframes.

The table below shows typical MIPS thresholds across small, medium, and large enterprise organizations (SORGs, MORGs, and LORGs).

CUSTOMER SIZE	TYPICAL MIPS USAGE
SORG	Less than 500 MIPS
MORG	500 MIPS to 5,000 MIPS
LORG	More than 5,000 MIPS

Mainframe data

Mainframe data is stored and organized in a variety of ways, from relational and hierarchical databases to high throughput file systems. Some of the common data systems are z/OS Db2 for relational data and IMS DB for hierarchical data. For high throughput file storage, you might see VSAM (IBM Virtual Storage Access Method). The following table provides a mapping of some of the more common mainframe data systems, and their possible migration targets into Azure.

DATA SOURCE	TARGET PLATFORM IN AZURE
z/OS Db2 & Db2 LUW	Azure SQL DB, SQL Server on Azure VMs, Db2 LUW on Azure VMs, Oracle on Azure VMs, Azure Database for PostgreSQL
IMS DB	Azure SQL DB, SQL Server on Azure VMs, Db2 LUW on Azure VMs, Oracle on Azure VMs, Azure Cosmos DB
Virtual Storage Access Method (VSAM), Indexed Sequential Access Method (ISAM), other flat files	Azure SQL DB, SQL Server on Azure VMs, Db2 LUW on Azure VMs, Oracle on Azure VMs, Azure Cosmos DB
Generation Date Groups (GDGs)	Files on Azure using extensions in the naming conventions to provide similar functionality to GDGs

Midrange systems, Unix variants, and other legacy systems

Midrange systems and midrange computers are loosely defined terms for a computer system that is more powerful than a general-purpose personal computer, but less powerful than a full-size mainframe computer. In most instances, a midrange computer is employed as a network server, when there are a small to medium

number of client systems. The computers generally have multiple processors, a large amount of random access memory (RAM), and large hard drives. Additionally, they usually contain hardware that allows for advanced networking, and ports for connecting to more business-oriented peripherals (such as large-scale data storage devices).

Common systems in this category include AS/400 and the IBM i and p series. Unisys also has a collection of midrange systems.

Unix operating system

The Unix operating system was one of the first enterprise-grade operating systems. Unix is the base operating system for Ubuntu, Solaris, and operating systems that follow POSIX standards. Unix was developed in the 1970s by Ken Thompson, Dennis Ritchie, and others at AT&T Laboratories. It was originally meant for programmers who are developing software, rather than non-programmers. It was distributed to government organizations and academic institutions, both of which led Unix to being ported to a wider variety of variations and forks, with different specialized functions. Unix and its variants (such as AIX, HP-UX, and Tru64) are commonly found running on legacy systems, such as IBM mainframes, AS/400 systems, Sun Sparc, and DEC hardware-based systems.

Other systems

Other legacy systems include the family of systems from Digital Equipment Corporation (DEC), such as the DEC VAX, DEC Alpha, and DEC PDP. The DEC systems initially ran the VAX VMS operating system, then eventually they moved to Unix variants, such as Tru64. Other systems include ones that are based on the PA-RISC architecture, such as the HP-3000 and HP-9000 systems.

Midrange data and storage

Midrange data is stored and organized in variety of ways, from relational and hierarchical databases, to high throughput file systems. Some of the common data systems are Db2 for i (for relational data), and IMS DB for hierarchical data. The following table provides a mapping of some of the more common mainframe data systems and the possible migration targets into Azure.

DATA SOURCE	TARGET PLATFORM IN AZURE
Db2 for i	Azure SQL DB, SQL Server on Azure VMs, Azure Database for PostgreSQL, Db2 LUW on Azure VMs, Oracle on Azure VMs
IMS DB	Azure SQL DB, SQL Server on Azure VMs, Db2 LUW on Azure VMs, Oracle on Azure VMs, Azure Cosmos DB

Endianness

Consider the following details about endianness:

- RISC and x86 processors differ in *endianness*, a term used to describe how a system stores bytes in computer memory.
- RISC-based computers are known as big endian systems, because they store the most significant ("big") value first—that is, in the lowest storage address.
- Most Linux computers are based on the x86 processor, which are little endian systems, meaning they store the least significant ("little") value) first.

The following figure visually shows you the difference between big endian and little endian.

High-level architectural types

Rehost

Often referred to as a lift-and-shift migration, this option doesn't require code changes. You can use it to quickly migrate your existing applications to Azure. Each application is migrated as is, to reap the benefits of the cloud (without the risk and cost that are associated with code changes).

Rehost architectures

- Stromasys Charon-SSP Solaris emulator on Azure VMs

  - 09/09/2020
  - 4 min read

  Charon-SSP cross-platform hypervisor emulates legacy Sun SPARC systems on industry standard x86-64 computer systems and VMs.

- Migrate IBM mainframe apps to Azure with TmaxSoft OpenFrame

  - 09/18/2020
  - 5 min read

  Migrate IBM zSeries mainframe applications to Azure. Use a no-code approach that TmaxSoft OpenFrame offers for this lift and shift operation.

- Unisys ClearPath Forward mainframe rehost to Azure using Unisys virtualization

  - 06/28/2021
  - 6 min read

- Using LzLabs Software Defined Mainframe (SDM) in an Azure VM deployment

  - 06/28/2021

- 7 min read

An approach for rehosting mainframe legacy applications in Azure using the LzLabs Software Defined Mainframe platform.

Refactor

Refactoring requires minimal changes to applications. This often enables the application architecture to leverage Azure platform as a service (PaaS) and to leverage additional cloud offerings. For example, you could migrate compute components of existing applications to Azure App Service or to Azure Kubernetes Service (AKS). You could also refactor relational and nonrelational databases into various options, such as Azure SQL Managed Instance, Azure Database for MySQL, Azure Database for PostgreSQL, and Azure Cosmos DB.

Refactor architectures

- General mainframe refactor to Azure

  - 08/10/2021
  - 8 min read

  See how to refactor general mainframe applications to run more cost-effectively and efficiently on Azure.

- Micro Focus Enterprise Server on Azure VMs

  - 03/01/2021
  - 6 min read

  Optimize, modernize, and streamline IBM z/OS mainframe applications by using Micro Focus Enterprise Server 6.0 on Azure VMs.

- Refactor IBM z/OS mainframe Coupling Facility (CF) to Azure

  - 12/01/2020
  - 7 min read

  Learn how Azure services and components can provide scale-out performance comparable to IBM z/OS mainframe CF and Parallel Sysplex capabilities.

- Unisys Dorado mainframe migration to Azure with Astadia & Micro Focus

  - 03/19/2021
  - 9 min read

  Migrate Unisys Dorado mainframe systems with Astadia and Micro Focus products. Move to Azure without rewriting code, switching data models, or updating screens.

- Unisys mainframe migration

- 11/17/2020
- 7 min read

Learn about options for using the Asysco Automated Migration Technology (AMT) Framework to migrate Unisys mainframe workloads to Azure.

- IBM System i (AS/400) to Azure using Infinite i

  - 05/14/2021
  - 7 min read

  Use Infinite i to easily migrate your IBM System i (AS/400) workloads to Azure. You can lower costs, improve performance, improve availability, and modernize.

- IBM z/OS mainframe migration with Asysco

  - 04/01/2021
  - 7 min read

  See how to use the Asysco Automated Migration Technology (AMT) framework to migrate IBM z/OS mainframe workloads to Azure.

- Refactor mainframe applications to Azure with Raincode compilers

  - 06/25/2021
  - 7 min read

  This architecture shows how the Raincode COBOL compiler modernizes mainframe legacy applications.

- IBM z/OS online transaction processing on Azure

  - 07/30/2021
  - 8 min read

  Migrate a z/OS online transaction processing (OLTP) workload to an Azure application that is cost-effective, responsive, scalable, and adaptable.

Re-engineer

Re-engineering for migration focuses on modifying and extending application functionality and the code base to optimize the application architecture for cloud scalability. For example, you could break down a monolithic application into a group of microservices that work together and scale easily. You could also rearchitect relational and nonrelational databases to a fully managed database solution, such as SQL Managed Instance, Azure Database for MySQL, Azure Database for PostgreSQL, and Azure Cosmos DB.

Re-engineer architectures

-

High-volume batch transaction processing

- 03/22/2021
- 6 min read

Use Azure Kubernetes Service (AKS) and Azure Service Bus to implement high-volume batch transaction processing.

- Integrate IBM mainframe and midrange message queues with Azure

  - 06/02/2021
  - 7 min read

  This example describes a data-first approach to middleware integration that enables IBM message queues (MQs).

- Re-engineer IBM z/OS batch applications on Azure

  - 8/13/2021
  - 9 min read

  Use Azure services to re-engineer mainframe batch applications. This architecture change can reduce costs and improve scalability.

Dedicated hardware

Another pattern for migrations into Azure (for legacy systems) is what is known as *dedicated hardware*. This pattern is where legacy hardware (such as IBM Power Systems) runs inside the Azure datacenter, with an Azure managed-service wrapping around the hardware, which enables easy cloud management and automation. Further, this hardware is available to connect to and use with other Azure IaaS and PaaS services.

Dedicated hardware architectures

- Migrate AIX workloads to Skytap on Azure

  - 05/24/2021
  - 5 min read

  This example illustrates a migration of AIX logical partitions (LPARs) to Skytap on Azure.

- Migrate IBM i series applications to Skytap on Azure

  - 05/24/2021
  - 5 min read

  This example architecture shows how to use the native IBM i backup and recovery services with Microsoft Azure components.

Data movement and migration

A key part of legacy migrations and transformations to Azure is consideration for data. This can include not only

data movement, but also data replication and synchronization.

Data movement and migration architectures

- Modernize mainframe & midrange data

  - 11/04/2020
  - 7 min read

  Learn how to modernize IBM mainframe and midrange data. See how to use a data-first approach to migrate this data to Azure.

- Replicate and sync mainframe data in Azure

  - 12/29/2020
  - 8 min read

  Replicate data while modernizing mainframe and midrange systems. Sync on-premises data with Azure data during modernization.

- Mainframe access to Azure databases

  - 05/03/2021
  - 4 min read

  Give mainframe applications access to Azure data without changing code. Use Microsoft Service for DRDA to run Db2 SQL statements on a SQL Server database.

- Mainframe file replication and sync on Azure

  - 04/01/2021
  - 5 min read

  Learn about several options for moving, converting, transforming, and storing mainframe and midrange file system data on-premises and in Azure.

Next steps

- For more information, please contact legacy2azure@microsoft.com.
- See the Microsoft Azure Well-Architected Framework.

Related resources

The white papers, blogs, webinars, and other resources are available to help you on your journey, to understand the pathways to migrate legacy systems into Azure:

Whitepapers

- Stromasys Charon-SSP Solaris Emulator: Azure Setup Guide

- Stromasys legacy server emulation on Azure: Running applications designed for SPARC, Alpha, VAX, PDP-11, and HP 3000
- Deploy Db2 pureScale on Azure (Whitepaper)
- Install IBM DB2 pureScale on Azure (Azure Docs)
- Demystifying mainframe to Azure migration
- Microsoft Azure Government cloud for mainframe applications
- Set up Micro Focus Enterprise Server 4.0 and Enterprise Developer 4.0 in Azure
- Set up IBM Z Development and Test Environment 12.0 in Azure
- Move mainframe compute and storage to Azure
- E-Book: Install TmaxSoft OpenFrame on Azure

Webinars

- Angelbeat - Retail Industry Legacy Webinar
- Mainframe Transformation to Azure
- Mainframe Transformation: Azure is the New Mainframe
- ClearPath MCP Software Series For Azure
- Leverage the Power of Azure with Steve Read
- Carahsoft - Monolithic Mainframe to Azure Gov Cloud The USAF Journey
- Carahsoft - Topics in Government Mainframe Transformation to Azure Gov Cloud
- Skytap on Azure Webinar
- Bridge to Application Modernization: Virtualized SPARC/PA-RISK/DEC to Azure

Blog posts

- Running Micro Focus Enterprise Server 4.0 in a Docker Container in Azure
- Deploy Micro Focus Enterprise Server 4.0 to AKS
- Migrating iSeries (AS/400) Legacy Applications to Azure
- Migrating iSeries (AS/400) Legacy Applications to Azure with Infinite
- Migrating AIX Workloads to Azure: Approaches and Best Practices
- Using Containers for Mainframe Modernization
- Deploying NTT Data UniKix in Azure, Part 1 Deploying the VM
- MIPS Equivalent Sizing for IBM CICS COBOL Applications Migrated to Microsoft Azure
- Set up Micro Focus Enterprise Server 4.0 and Enterprise Developer 4.0 in Azure
- Set up IBM Z Development and Test Environment 12.0 in Azure

Customer stories

Different industries are migrating from legacy mainframe and midrange systems in innovative and inspiring ways. Following are a number of customer case studies and success stories:

- Mainframe to Azure: A Real World Modernization Case Study (GEICO and AIS)
- Jefferson County, Alabama
- Customer Technical Story: Actuarial Services Company - DEC Alpha to Azure using Stromasys
- Astadia & USAF Complete Mission-Critical Mainframe-to-Cloud Migration | Business Wire
- United States Air Force | Case Study (astadia.com)

Add IP address spaces to peered virtual networks

11/19/2021 • 3 minutes to read • Edit Online

Many organizations deploy a virtual networking architecture that follows the hub-spoke model. At some point, the hub virtual network might require additional IP address spaces. However, address ranges can't be added or deleted from a virtual network's address space once it's peered with another virtual network. To add or remove address ranges, delete the peering, add or remove the address ranges, then re-create the peering manually. The scripts described in this article can make that process easier.

> **NOTE**
>
> This article has not yet been updated to reflect Azure networking's support for peering resync. Azure virtual networks support adding and removing address space without the need to remove and restablish peerings; instead each remote peering needs a sync operation performed after the network space has changed. The sync can be performed using the Sync-AzVirtualNetworkPeering PowerShell command or from the Azure Portal.

Single subscription

A single subscription use case, both hub and all spoke virtual networks are in the same subscription.

Multiple subscriptions

Another use case can be where the hub virtual network is in one subscription and all other spoke virtual networks are in different subscriptions. The subscriptions all share a single Azure Active Directory tenant.

Considerations

- Running the script will result in outage or disconnections between the hub and spoke virtual networks. Execute it during an approved maintenance window.
- Run `Get-Module -ListAvailable Az` to find the installed version. The script requires the Azure PowerShell module version 1.0.0 or later. If you need to upgrade, see Install Azure PowerShell module.
- If not already connected, run `Connect-AzAccount` to create a connection with Azure.
- Consider assigning accounts, used for virtual network peering, to the Network Contributor role or a custom role containing the necessary actions found under virtual network peering permissions.
- Assign accounts used to add IP address spaces, to the Network Contributor role or a custom role containing the necessary actions found under virtual network permissions.
- The IP address space that you want to add to the hub virtual network must not overlap with any of the IP address spaces of the spoke virtual networks that you intend to peer with the hub virtual network.

Add the IP address range

The script automatically removes all Virtual Network peerings from the hub Virtual Network, adds an IP address range prefix to the hub Virtual Network based on input parameters, adds the Virtual Network peerings back to the hub Virtual Network, and reconnects the hub Virtual Network peerings to the existing spoke Virtual Network peerings. The script applies to single and multiple subscription hub-spoke topologies.

```
param (
    # Address Prefix range (CIDR Notation, e.g., 10.0.0.0/24 or 2607:f000:0000:00::/64)
    [Parameter(Mandatory = $true)]
    [String[]]
    $IPAddressRange,

    # Hub VNet Subscription Name
    [Parameter(Mandatory = $true)]
    [String]
```

```
    [String]
    $HubVNetSubscriptionName,

    # Hub VNet Resource Group Name
    [Parameter(Mandatory = $true)]
    [String]
    $HubVNetRGName,

    # Hub VNet Name
    [Parameter(Mandatory = $true)]
    [String]
    $HubVNetName
)

#Set context to Hub VNet Subscription
Get-AzSubscription -SubscriptionName $HubVNetSubscriptionName | Set-AzContext
#end

#Get All Hub VNet Peerings and Hub VNet Object
$hubPeerings = Get-AzVirtualNetworkPeering -ResourceGroupName $HubVNetRGName -VirtualNetworkName
$HubVNetName
$hubVNet = Get-AzVirtualNetwork -Name $HubVNetName -ResourceGroupName $HubVNetRGName
#end

#Remove All Hub VNet Peerings
Remove-AzVirtualNetworkPeering -VirtualNetworkName $HubVNetName -ResourceGroupName $HubVNetRGName -name
$hubPeerings.Name -Force
#end

#Add IP address range to the hub vnet
$hubVNet.AddressSpace.AddressPrefixes.Add($IPAddressRange)
#end

#Apply configuration stored in $hubVnet
Set-AzVirtualNetwork -VirtualNetwork $hubVNet
#end

foreach ($vNetPeering in $hubPeerings)
{
    # Get remote vnet name
    $vNetFullId = $vNetPeering.RemoteVirtualNetwork.Id
    $vNetName = $vNetFullId.Substring($vNetFullId.LastIndexOf('/') + 1)

    # Pull remote vNet object
    $vNetObj = Get-AzVirtualNetwork -Name $vNetName

    # Get the peering from the remote vnet object
    $peeringName = $vNetObj.VirtualNetworkPeerings.Where({$_.RemoteVirtualNetwork.Id -like
"*$($hubVNet.Name)"}).Name
    $peering = Get-AzVirtualNetworkPeering -ResourceGroupName $vNetObj.ResourceGroupName -VirtualNetworkName
$vNetName -Name $peeringName

    # Reset to initiated state
    Set-AzVirtualNetworkPeering -VirtualNetworkPeering $peering

    # Re-create peering on hub
    Add-AzVirtualNetworkPeering -Name $vNetPeering.Name -VirtualNetwork $HubVNet -RemoteVirtualNetworkId
$vNetFullId -AllowGatewayTransit
}
```

Pricing

There is a nominal charge for ingress and egress traffic that utilizes a virtual network peering. There is no change to existing pricing when adding an additional IP address space to an Azure Virtual Network. For more information, see the pricing page.

Next steps

- Learn more about managing Virtual Network peerings.
- Learn more about managing IP Address ranges on Virtual Networks.

Azure Well-Architected Framework review of Azure Application Gateway

11/19/2021 • 14 minutes to read • Edit Online

This article provides architectural best practices for the Azure Application Gateway v2 family of SKUs. The guidance is based on the five pillars of architecture excellence: Cost Optimization, Operational Excellence, Performance Efficiency, Reliability, and Security.

We assume that you have working knowledge of Azure Application Gateway and are well versed with v2 SKU features. As a refresher, review the full set of Azure Application Gateway features.

Cost Optimization

Review and apply the cost principles when making design choices. Here are some best practices.

Review Application Gateway pricing

Familiarize yourself with Application Gateway pricing to help you identify the right deployment configuration for your environment. Ensure that the options are adequately sized to meet the capacity demand and deliver expected performance without wasting resources.

For information about Application Gateway pricing, see Understanding Pricing for Azure Application Gateway and Web Application Firewall.

Use these resources to estimate cost based on units of consumption.

- Azure Application Gateway pricing
- Pricing calculator

Review underutilized resources

Identify and delete Application Gateway instances with empty backend pools.

Stop Application Gateway instances when not in use

You aren't billed when Application Gateway is in the stopped state.

Continuously running Application Gateway instances can incur extraneous costs. Evaluate usage patterns and stop instances when you don't need them. For example, usage after business hours in Dev/Test environments is expected to be low.

See these articles for information about how to stop and start instances.

- Stop-AzApplicationGateway
- Start-AzApplicationGateway

Have a scale-in and scale-out policy

A scale-out policy ensures that there will be enough instances to handle incoming traffic and spikes. Also, have a scale-in policy that makes sure the number of instances are reduced when demand drops. Consider the choice of instance size. The size can significantly impact the cost. Some considerations are described in the Estimate the Application Gateway instance count.

For more information, see Autoscaling and Zone-redundant Application Gateway v2.

Review consumption metrics across different parameters

You're billed based on metered instances of Application Gateway based on the metrics tracked by Azure. Here's an example of cost incurred view in Azure Cost Management + Billing.

> The example is based on the current price and is subject to change. This is shown for information purposes only.

Scope : rg-bu0001a0008 | VIEW Cost by resource | Jun 2021 | Add filter

ACTUAL COST (USD) $82.55 | FORECAST: CHART VIEW ON -- | BUDGET: NONE --

Filter items | 10 rows

Resource	Resource type	Location	Resource group name
apw-aks-u4lhb2tam5j4s	Application gateway	us east 2	rg-bu0001a0008

Service name	Meter	Cost
application gateway	fixed cost	$40.60
application gateway	capacity units	$0.83
bandwidth	data transfer out	<$0.01

la-aks-u4lhb2tam5j4s	Log Analytics workspace	us east 2	rg-bu0001a0008
acraksu4lhb2tam5j4s	Container registry	us east 2	rg-bu0001a0008
nodepools-to-akv	Private endpoint	us east 2	rg-bu0001a0008
nodepool-to-acr	Private endpoint	us east 2	rg-bu0001a0008

Evaluate the various metrics and capacity units and determine the cost drivers.

These are key metrics for Application Gateway. This information can be used to validate that the provisioned instance count matches the amount of incoming traffic.

- **Estimated Billed Capacity Units**
- **Fixed Billable Capacity Units**
- **Current Capacity Units**

For more information, see Application Gateway metrics.

Make sure you account for bandwidth costs. For details, see Traffic across billing zones and regions.

Performance Efficiency

Take advantage features for autoscaling and performance benefits

The v2 SKU offers autoscaling to ensure that your Application Gateway can scale up as traffic increases. When compared to v1 SKU, v2 has capabilities that enhance the performance of the workload. For example, better TLS offload performance, quicker deployment and update times, zone redundancy, and more. For more information about autoscaling features, see Autoscaling and Zone-redundant Application Gateway v2.

If you are running v1 SKU gateways, consider migrating to v2 SKU. See
Migrate Azure Application Gateway and Web Application Firewall from v1 to v2.

General best practices related to Performance Efficiency are described in Performance efficiency principles.

Estimate the Application Gateway instance count

Application Gateway v2 scales out based on many aspects, such as CPU, memory, network utilization, and more. To determine the approximate instance count, factor in these metrics:

- **Current compute units**—Indicates CPU utilization. 1 Application Gateway instance is approximately 10 compute units.

- **Throughput**—Application Gateway instance can serve 60-75 Mbps of throughput. This data depends on the type of payload.

Consider this equation when calculating instance counts.

$$\text{Approximate instance count} = \max\left(\frac{\text{Current compute units}}{10}, \frac{\text{Throughput in Mbps}}{60}\right)$$

After you've estimated the instance count, compare that value to the maximum instance count. This will indicate how close you are to the maximum available capacity.

Define the minimum instance count

For Application Gateway v2 SKU, autoscaling takes some time (approximately six to seven minutes) before the additional set of instances is ready to serve traffic. During that time, if there are short spikes in traffic, expect transient latency or loss of traffic.

We recommend that you set your minimum instance count to an optimal level. After you estimate the average instance count and determine your Application Gateway autoscaling trends, define the minimum instance count based on your application patterns. For information, see Application Gateway high traffic support.

Check the **Current Compute Units** for the past one month. This metric represents the gateway's CPU utilization. To define the minimum instance count, divide the peak usage by 10. For example, if your average **Current Compute Units** in the past month is 50, set the minimum instance count to 5.

Define the maximum instance count

We recommend 125 as the maximum autoscale instance count. Make sure the subnet that has the Application Gateway has sufficient available IP addresses to support the scale-up set of instances.

Setting the maximum instance count to 125 has no cost implications because you're billed only for the consumed capacity.

Define Application Gateway subnet size

Application Gateway needs a dedicated subnet within a virtual network. The subnet can have multiple instances of the deployed Application Gateway resource. You can also deploy other Application Gateway resources in that subnet, v1 or v2 SKU.

Here are some considerations for defining the subnet size:

- Application Gateway uses one private IP address per instance and another private IP address if a private front-end IP is configured.
- Azure reserves five IP addresses in each subnet for internal use.
- Application Gateway (Standard or WAF SKU) can support up to 32 instances. Taking 32 instance IP addresses + 1 private front-end IP + 5 Azure reserved, a minimum subnet size of /26 is recommended. Because the Standard_v2 or WAF_v2 SKU can support up to 125 instances, using the same calculation, a subnet size of /24 is recommended.
- If you want to deploy additional Application Gateway resources in the same subnet, consider the additional IP addresses that will be required for their maximum instance count for both, Standard and Standard v2.

Operational Excellence

Monitoring and diagnostics are crucial. Not only can you measure performance statistics but also use metrics troubleshoot and remediate issues quickly.

Monitor capacity metrics

Use these metrics as indicators of utilization of the provisioned Application Gateway capacity. We strongly

recommend setting up alerts on capacity. For details, see Application Gateway high traffic support.

METRIC	DESCRIPTION	USE CASE
Current Compute Units	CPU utilization of virtual machine running Application Gateway. One Application Gateway instance supports 10 Compute Units.	Helps detect issues when more traffic is sent than what Application Gateway instances can handle.
Throughput	Amount of traffic (in Bps) served by Application Gateway.	This threshold is dependent on the payload size. For smaller payloads but more frequent connections, expect lower throughput limits and adjust alerts accordingly.
Current Connections	Active TCP connections on Application Gateway.	Helps detect issues where the connection count increases beyond the capacity of Application gateway. Look for a drop in capacity unit when the connection count increases, look for a simultaneous drop in capacity unit. This will indicate if Application Gateway is out of capacity.

Troubleshoot using metrics

There are other metrics that can indicate issues either at Application Gateway or the backend. We recommend evaluating alerts as per the table below.

METRIC	DESCRIPTION	USE CASE
Unhealthy Host Count	Number of backends that Application Gateway is unable to probe successfully.	Application Gateway instances are unable to connect to the backend. For example, the probe interval is 10 seconds and unhealthy host count threshold is 3 failed probes). A backend will turn unhealthy if Application Gateway instance isn't able to reach it for 30 seconds. Also depends on the configured timeout and interval in the custom probe.
Response Status (dimension 4xx and 5xx)	The HTTP response status returned to clients from Application Gateway. This status is usually same as the **Backend Response Status**, unless Application Gateway is unable to get a response from the backend or Application Gateway has an internal error in serving responses.	Issues with Application Gateway or the backend. Use this metric with **Backend Response Status** to identify whether Application Gateway or the backend is failing to serve requests.
Backend Response Status (dimension 4xx and 5xx)	The HTTP response status returned to Application Gateway from the backend.	Use to validate if the backend is successfully receiving requests and serving responses.
Backend Last Byte Response Time	Time interval between the start of a connection to backend server and receiving the last byte of the response body.	Increase in this latency implies that the backend is getting loaded and is taking longer to respond to requests. One way to resolve this issue is to scale up the backend.

METRIC	DESCRIPTION	USE CASE
Application Gateway Total Time	Time period from when Application Gateway receives the first byte of the HTTP request to when the last response byte has been sent to the client. This includes client RTT	Increase in this latency, without any accompanying application changes or access traffic pattern changes should be investigated. If this metric increases, monitor other the metrics and determine if they other metrics are also increasing, such as compute units, total throughput, or total request count.

Enable diagnostics on Application Gateway and web application firewall (WAF)

Diagnostic logs allow you to view firewall logs, performance logs, and access logs. Use these logs to manage and troubleshoot issues with Application Gateway instances.

Use Azure Monitor Network Insights

Azure Monitor Network Insights provides a comprehensive view of health and metrics for network resources, including Application Gateway. For additional details and supported capabilities for Application Gateway, see Azure Monitor Network insights.

Use advanced monitoring metrics

Consider monitoring and setting alerts on metrics such as **Unhealthy host count**, and metrics that indicate the latency and the number of connections and requests. Notice the difference between connections and requests for Application Gateway frontend connections. One connection represents the TCP connection (sockets pair), while a request represents a resource request, that is a GET, PUT, POST, and so on. One connection can serve multiple requests (on Application Gateway v2, up to 100).

SNAT port limitations

SNAT port limitations are important for backend connections on the Application Gateway. There are separate factors that affect how Application Gateway reaches the SNAT port limit. For example, if the backend is a public IP, it will require its own SNAT port. In order to avoid SNAT port limitations, you can increase the number of instances per Application Gateway, scale out the backends to have more IPs, or move your backends into the same virtual network and use private IP addresses for the backends.

Requests per second (RPS) considerations

Requests per second (RPS) on the Application Gateway will be affected if the SNAT port limit is reached. For example, if Application Gateway has reached the SNAT port limit, then Application Gateway won't be able to open a new connection to the backend and the request will fail.

Match timeout settings with the backend application

Ensure you have configured the **IdleTimeout** settings to match the listener and traffic characteristics of the backend application. The default value is set to 4 minutes and can be configured to a maximum of 30. For more information, see Load Balancer TCP Reset and Idle Timeout.

For workload considerations, see Application Monitoring.

Monitoring Key Vault configuration issues through Azure Advisor

Azure Application Gateway checks for the renewed certificate version in the linked Key Vault at every 4-hour interval. If it is inaccessible due to any incorrectly modified Key Vault configurations, it logs that error and pushes a corresponding Advisor recommendation. You must configure Advisor alert to stay updated and fix such issues immediately to avoid any Control or Data plane related problems. To set an alert for this specific case, use the Recommendation Type as "Resolve Azure Key Vault issue for your Application Gateway".

Reliability

Here are some best practices to minimize failed instances.

In addition, we recommend that you review the Principles of the reliability pillar.

Plan for rule updates

Plan enough time for updates before accessing Application Gateway or making further changes. For example, removing servers from backend pool might take some time because they have to drain existing connections.

Use health probes to detect backend unavailability

If Application Gateway is used to load balance incoming traffic over multiple backend instances, we recommend the use of health probes. These will ensure that traffic is not routed to backends that are unable to handle the traffic.

Review the impact of the interval and threshold settings on health probes

The health probe sends requests to the configured endpoint at a set *interval*. Also, there's a *threshold* of failed requests that will be tolerated before the backend is marked unhealthy. These numbers present a trade-off.

- Setting a higher interval puts a higher load on your service. Each Application Gateway instance sends its own health probes, so 100 instances every 30 seconds means 100 requests per 30 seconds.
- Setting a lower interval leaves more time before an outage is detected.
- Setting a low unhealthy threshold may mean that short, transient failures may take down a backend.
- Setting a high threshold it can take longer to take a backend out of rotation.

Verify downstream dependencies through health endpoints

Suppose each backend has its own dependencies to ensure failures are isolated. For example, an application hosted behind Application Gateway may have multiple backends, each connected to a different database (replica). When such a dependency fails, the application may be working but won't return valid results. For that reason, the health endpoint should ideally validate all dependencies. Keep in mind that if each call to the health endpoint has a direct dependency call, that database would receive 100 queries every 30 seconds instead of 1. To avoid this, the health endpoint should cache the state of the dependencies for a short period of time.

For more information, see these articles:

- Health monitoring overview for Azure Application Gateway
- Azure Front Door - backend health monitoring
- Health probes to scale and provide HA for your service

Security

Security is one of the most important aspects of any architecture. Application Gateway provides features to employ both the principle of least privilege and defense-in-defense. We recommend you also review the Security design principles.

Restrictions of Network Security Groups (NSGs)

NSGs are supported on Application Gateway, but there are some restrictions. For instance, some communication with certain port ranges is prohibited. Make sure you understand the implications of those restrictions. For details, see Network security groups.

User Defined Routes (UDR)-supported scenarios

Using User Defined Routes (UDR) on the Application Gateway subnet cause some issues. Health status in the back-end might be unknown. Application Gateway logs and metrics might not get generated. We recommend that you don't use UDRs on the Application Gateway subnet so that you can view the back-end health, logs, and metrics. If your organizations require to use UDR in the Application Gateway subnet, please ensure you review the supported scenarios. For details, see Supported user-defined routes.

DNS lookups on App Gateway subnet

When the backend pool contains a resolvable FQDN, the DNS resolution is based on a private DNS zone or custom DNS server (if configured on the VNet), or it uses the default Azure-provided DNS.

Set up a TLS policy for enhanced security

Set up a TLS policy for extra security. Ensure you're using the latest TLS policy version (AppGwSslPolicy20170401S). This enforces TLS 1.2 and stronger ciphers.

Use AppGateway for TLS termination

There are advantages of using Application Gateway for TLS termination:

- Performance improves because requests going to different backends to have to re-authenticate to each backend.
- Better utilization of backend servers because they don't have to perform TLS processing
- Intelligent routing by accessing the request content.
- Easier certificate management because the certificate only needs to be installed on Application Gateway.

Encrypting considerations

When re-encrypting backend traffic, ensure the backend server certificate contains both the root and intermediate Certificate Authorities (CAs). A TLS certificate of the backend server must be issued by a well-known CA. If the certificate was not issued by a trusted CA, the Application Gateway checks if the certificate of the issuing CA was issued by a trusted CA, and so on until either a trusted CA is found. Only then a secure connection is established. Otherwise, Application Gateway marks the backend as unhealthy.

Azure Key Vault for storing TLS certificates

Application Gateway is integrated with Key Vault. This provides stronger security, easier separation of roles and responsibilities, support for managed certificates, and an easier certificate renewal and rotation process.

Enabling the Web Application Firewall (WAF)

When WAF is enabled, every request must be buffered by the Application Gateway until it fully arrives and check if the request matches with any rule violation in its core rule set and then forward the packet to the backend instances. For large file uploads (30MB+ in size), this can result in a significant latency. Because Application Gateway capacity requirements are different with WAF, we do not recommend enabling WAF on Application Gateway without proper testing and validation.

Next steps

Microsoft Azure Well-Architected Framework

Azure Well-Architected Framework review of Azure Firewall

11/19/2021 • 13 minutes to read • Edit Online

This article provides architectural best practices for Azure Firewall. The guidance is based on the five pillars of architecture excellence: cost optimization, operational excellence, performance efficiency, reliability, and security.

Cost optimization

Review underutilized Azure Firewall instances, and identify and delete Azure Firewall deployments not in use. To identify Azure Firewall deployments not in use, start analyzing the Monitoring Metrics and User Defined Routes (UDRs) that are associated with subnets pointing to the Firewall's private IP. Then, combine that with additional validations, such as if the Azure Firewall has any Rules (Classic) for NAT, or Network and Application, or even if the DNS Proxy setting is configured to **Disabled**, as well as with internal documentation about your environment and deployments. See the details about monitoring logs and metrics at Monitor Azure Firewall logs and metrics and SNAT port utilization.

Share the same Azure Firewall across multiple workloads and Azure Virtual Networks. Deploy a central Azure Firewall in the hub virtual network, and share the same Firewall across many spoke virtual networks that are connected to the same hub from the same region. Ensure that there is no unexpected cross-region traffic as part of the hub-spoke topology.

Stop Azure Firewall deployments that do not need to run for 24 hours. This could be the case for development environments that are used only during business hours. See more details at Deallocate and allocate Azure Firewall.

Properly size the number of Public IPs that your firewall needs. Validate whether all the associated Public IPs are in use. If they are not in use, disassociate and delete them. Use IP Groups to reduce your management overhead. Evaluate SNAT ports utilization before you remove any IP Addresses. See the details about monitoring logs and metrics at Monitor Azure Firewall logs and metrics and SNAT port utilization.

Use Azure Firewall Manager and its policies to reduce your operational costs, by increasing the efficiency and reducing your management overhead. Review your Firewall Manager policies, associations, and inheritance carefully. Policies are billed based on firewall associations. A policy with zero or one firewall association is free of charge. A policy with multiple firewall associations is billed at a fixed rate. See more details at Pricing - Firewall Manager.

Review the differences between the two Azure Firewall SKUs. The Standard option is usually enough for east-west traffic, where Premium comes with the necessary additional features for north-south traffic, as well as the forced tunneling feature and many other features. See more information at Azure Firewall Premium Preview features. Deploy mixed scenarios using the Standard and Premium options, according to your needs.

Operational excellence

General administration and governance

- Use Azure Firewall to govern:
  - Internet outbound traffic (VMs and services that access the internet)
  - Non-HTTP/S inbound traffic
  - East-west traffic filtering

- Use Azure Firewall Premium, if any of the following capabilities are required:
  - TLS inspection - Decrypts outbound traffic, processes the data, encrypts the data, and then sends it to the destination.
  - IDPS - A network intrusion detection and prevention system (IDPS) allows you to monitor network activities for malicious activity, log information about this activity, report it, and optionally attempt to block it.
  - URL filtering - Extends Azure Firewall's FQDN filtering capability to consider an entire URL. For example, the filtered URL might be www.contoso.com/a/c instead of www.contoso.com.
  - Web categories - Administrators can allow or deny user access to website categories, such as gambling websites, social media websites, and others.
  - See more details at Azure Firewall Premium Preview features.
- Use Firewall Manager to deploy and manage multiple Azure Firewalls across Azure Virtual WAN hubs and hub-spoke based deployments.
- Create a global Azure Firewall policy to govern the security posture across the global network environment, and then assign it to all Azure Firewall instances. This allows for granular policies to meet the requirements of specific regions, by delegating incremental Azure Firewall policies to local security teams, via RBAC.
- Configure supported 3rd-party SaaS security providers within Firewall Manager, if you want to use such solutions to protect outbound connections.
- For existing deployments, migrate Azure Firewall rules to Azure Firewall Manager policies, and use Azure Firewall Manager to centrally manage your firewalls and policies.

Infrastructure provisioning and changes

- We recommend the Azure Firewall to be deployed in the hub VNet. Very specific scenarios might require additional Azure Firewall deployments in spoke virtual networks, but that is not common.
- Prefer using IP prefixes.
- Become familiar with the limits and limitations, especially SNAT ports. Do not exceed limits, and be aware of the limitations. See the Azure Firewall limits at Azure subscription limits and quotas - Azure Resource Manager. Also, learn more about any existing usability limitations at Azure Firewall FAQ.
- For concurrent deployments, make sure to use IP Groups, policies, and firewalls that do not have concurrent Put operations on them. Ensure all updates to the IP Groups and policies have an implicit firewall update that is run afterwards.
- Ensure a developer and test environment to validate firewall changes.
- A well-architected solution also involves considering the placement of your resources, to align with all functional and non-functional requirements. Azure Firewall, Application Gateway, and Load Balancers can be combined in multiple ways to achieve different goals. You can find scenarios with detailed recommendations, at Firewall and Application Gateway for virtual networks.

Networking

An Azure Firewall is a dedicated deployment in your virtual network. Within your virtual network, a dedicated subnet is required for the Azure Firewall. Azure Firewall will provision more capacity as it scales. A /26 address space for its subnets ensures that the firewall has enough IP addresses available to accommodate the scaling. Azure Firewall does not need a subnet bigger than /26, and the Azure Firewall subnet name must be `AzureFirewallSubnet`.

- If you are considering using the forced tunneling feature, you will need an additional /26 address space for the Azure Firewall Management subnet, and you must name it `AzureFirewallManagementSubnet` (this is also a requirement).
- Azure Firewall always starts with two instances, it can scale up to 20 instances, and you cannot see those individual instances. You can only deploy a single Azure Firewall instance in each VNet.
- Azure Firewall must have direct Internet connectivity. If your `AzureFirewallSubnet` learns a default route to your on-premises network via BGP, then you must configure Azure Firewall in the forced tunneling mode. If

this is an existing Azure Firewall instance, which cannot be reconfigured in the forced tunneling mode, then we recommended that you create a UDR with a 0.0.0.0/0 route, with the **NextHopType** value set as **Internet**. Associate it with the **AzureFirewallSubnet** to maintain internet connectivity.

- When deploying a new Azure Firewall instance, if you enable the forced tunneling mode, you can set the Public IP Address to **None** to deploy a fully private data plane. However, the management plane still requires a public IP, for management purposes only. The internal traffic from Virtual Networks, and/or on-premises, will not use that public IP. See more about forced tunneling at Azure Firewall forced tunneling.
- When you have multi-region Azure environments, remember that Azure Firewall is a regional service. Therefore, you'll likely have one instance per regional hub.

Monitoring

Monitoring capacity metrics

The following metrics can be used by the customer, as indicators of utilization of provisioned Azure Firewall capacity. Alerts can be set as needed by the customers, to get notifications once a threshold has been reached for any metric.

METRIC NAME	EXPLANATION
Application rules hit count	The number of times an application rule has been hit. Unit: count
Data processed	Sum of data traversing the firewall in a given time window. Unit: bytes
Firewall health state	Indicates the health of the firewall, based on SNAT port availability. Unit: percent This metric has two dimensions: - Status: Possible values are Healthy, Degraded, and Unhealthy. - Reason: Indicates the reason for the corresponding status of the firewall. If the SNAT ports are used > 95%, they are considered exhausted, and the health is 50% with status=Degraded and reason=SNAT port. The firewall keeps processing traffic, and the existing connections are not affected. However, new connections may not be established intermittently. If the SNAT ports are used < 95%, then the firewall is considered healthy, and the health is shown as 100%. If no SNAT ports usage is reported, then the health is shown as 0%.
Network rules hit count	The number of times a network rule has been hit. Unit: count

METRIC NAME	EXPLANATION
SNAT port utilization	The percentage of SNAT ports that have been utilized by the firewall. Unit: percent When you add more public IP addresses to your firewall, then more SNAT ports are available. This reduces the SNAT ports utilization. Additionally, when the firewall scales out for different reasons (for example, CPU or throughput), then additional SNAT ports also become available. Effectively, a given percentage of SNAT ports utilization may go down without you adding any public IP addresses, just because the service scaled out. You can directly control the number of public IP addresses that are available, to increase the ports available on your firewall. But, you can't directly control firewall scaling. If your firewall is running into SNAT port exhaustion, you should add at least five public IP address. This increases the number of SNAT ports available. Another option is to associate a NAT Gateway with the Azure Firewall subnet, which can help you increase the ports up to +1M ports.
Throughput	Rate of data traversing the firewall, per second. Unit: bits per second

Monitoring logs using Azure Firewall Workbook

Azure Firewall exposes a few other logs and metrics for troubleshooting that can be used as indicators of issues. We recommend evaluating alerts, as per the table below. Refer to Monitor Azure Firewall logs and metrics.

METRIC NAME	EXPLANATION
Application rule log	Each new connection that matches one of your configured application rules will result in a log for the accepted/denied connection.
Network rule log	Each new connection that matches one of your configured network rules will result in a log for the accepted/denied connection.
DNS Proxy log	This log tracks DNS messages to a DNS server that is configured using a DNS proxy.

Diagnostics logs and policy analytics

- Diagnostic logs allow you to view Azure Firewall logs, performance logs, and access logs. You can use these logs in Azure to manage and troubleshoot your Azure Firewall instance.
- Policy analytics for Azure Firewall Manager allows you to start seeing rules and flows that match the rules and hit count for those rules. By watching what rule is in use and the traffic that's being matched, you can have full visibility of the traffic.

Performance efficiency

SNAT ports exhaustion

- If more than 512K ports are necessary, use a NAT gateway with Azure Firewall. To scale up that limit, you can have up to +1M ports when associating a NAT gateway to the Azure Firewall subnet. For more information, refer to Scale SNAT ports with Azure NAT Gateway.

Auto scale and performance

- Azure Firewall uses auto scale. It can go up to 20 instances that provide up to 20 Gbps.
- Azure Firewall always starts with 2 instances. It scales up and down, based on CPU and the network throughput. After an auto scale, Azure Firewall ends up with either n-1 or n+1 instances.
- Scaling up happens if the threshold for CPU or throughput are greater than 60%, for more than five minutes.
- Scaling down happens if the threshold for CPU or throughput are under 60%, for more than 30 minutes. The scale-down process happens gracefully (deleting instances). The active connections on the deprovisioned instances are disconnected and switched over to other instances. For the majority of applications, this process does not cause any downtime, but applications should have some type of auto-reconnect capability. (The majority already has this capability.)
- If you're performing load tests, make sure to create initial traffic that is not part of your load tests, 20 minutes prior to the test. This is to allow the Azure Firewall instance to scale up its instances to the maximum. Use diagnostics settings to capture scale-up and scale-down events.
- Do not exceed 10k network rules, and make sure you use IP Groups. When creating network rules, remember that for each rule, Azure actually multiples **Ports x IP Addresses**, so if you have one rule with four IP address ranges and five ports, you will be actually consuming 20 network rules. Always try to summarize IP ranges.
- There are no restrictions for Application Rules.
- Add the Allow rules first, and then add the Deny rules to the lowest priority levels.

Reliability

- Azure Firewall provides different SLAs for when it is deployed in a single Availability Zone and for when it is deployed in multi-zones. For more information, see SLA for Azure Firewall. For information about all Azure SLAs, see the Azure service level agreements page.
- For workloads designed to be resistant to failures and to be fault-tolerant, remember to take into consideration that Azure Firewalls and Virtual Networks are regional resources.
- Closely monitor metrics, especially SNAT port utilization, firewall health state, and throughput.
- Avoid adding multiple individual IP addresses or IP address ranges to your network rules. Use super nets instead, or IP Groups when possible. Azure Firewall multiples **IPs x rules**, and that can make you reach the 10k recommended rules limit.

Security

- Understand rule processing logic:
  - Azure Firewall has NAT rules, network rules, and applications rules. The rules are processed according to the rule type. See more at Azure Firewall rule processing logic and Azure Firewall Manager rule processing logic.
- Use FQDN filtering in network rules.
  - You can use FQDNs in network rules, based on DNS resolution in Azure Firewall and Firewall policy. This capability allows you to filter outbound traffic with any TCP/UDP protocol (including NTP, SSH, RDP, and more). You must enable the DNS proxy to use FQDNs in your network rules. See how it works at Azure Firewall FQDN filtering in network rules.
- If you're filtering inbound Internet traffic with Azure Firewall policy DNAT, for security reasons, then the recommended approach is to add a specific Internet source, to allow DNAT access to the network and to avoid using wildcards.
- Use Azure Firewall to secure private endpoints (the virtual WAN scenario). See more at Secure traffic destined to private endpoints in Azure Virtual WAN.
- Configure threat intelligence:

    - Threat-intelligence-based filtering can be configured for your Azure Firewall policy to alert and deny traffic from and to known malicious IP addresses and domains. See more at Azure Firewall threat intelligence configuration.
- Use Azure Firewall Manager:
    - Azure Firewall Manager is a security management service that provides a central security policy and route management for cloud-based security perimeters. It includes the following features:
        - Central Azure Firewall deployment and configuration.
        - Hierarchical policies (global and local).
        - Integrated with third-party security-as-a-service for advanced security.
        - Centralized route management.
    - Understand how Policies are applied, at Azure Firewall Manager policy overview.
    - Use Azure Firewall policy to define a rule hierarchy. See Use Azure Firewall policy to define a rule hierarchy.
- Use Azure Firewall Premium:
    - Azure Firewall Premium is a next-generation firewall, with capabilities that are required for highly sensitive and regulated environments. It includes the following features:
        - TLS inspection - Decrypts outbound traffic, processes the data, encrypts the data, and then sends it to the destination.
        - IDPS - A network intrusion detection and prevention system (IDPS) allows you to monitor network activities for malicious activity, log information about this activity, report it, and optionally attempt to block it.
        - URL filtering - Extends Azure Firewall's FQDN filtering capability to consider an entire URL. For example, the filtered URL might be www.contoso.com/a/c instead of www.contoso.com.
        - Web categories - Administrators can allow or deny user access to website categories, such as gambling websites, social media websites, and others.
    - See more at Azure Firewall Premium Preview features.
- Deploy a security partner provider:
    - Security partner providers, in Azure Firewall Manager, allow you to use your familiar, best-in-breed, third-party security as a service (SECaaS) offering to protect Internet access for your users.
    - With a quick configuration, you can secure a hub with a supported security partner. You can route and filter Internet traffic from your Virtual Networks (VNets) or branch locations within a region. You can do this with automated route management, without setting up and managing user-defined routes (UDRs).
    - The current supported security partners are Zscaler, Check Point, and iboss.
    - See more at Deploy an Azure Firewall Manager security partner provider.

Next steps

- See the Microsoft Azure Well-Architected Framework.
- What is Azure Firewall?

Related resources

- Azure Firewall architecture overview
- Azure Well-Architected Framework review of Azure Application Gateway
- Firewall and Application Gateway for virtual networks
- Choose between virtual network peering and VPN gateways
- Hub-spoke network topology in Azure
- Security considerations for highly sensitive IaaS apps in Azure

Azure Well-Architected Framework review of an Azure NAT gateway

11/19/2021 • 6 minutes to read • Edit Online

This article provides best practices for an Azure NAT gateway. The guidance is based on the five pillars of architecture excellence: Cost Optimization, Operational Excellence, Performance Efficiency, Reliability, and Security.

We assume that you have a working knowledge of Azure Virtual Network NAT and Azure NAT gateway and that you are well-versed with the respective features. As a refresher, review the full set of Azure Virtual Network NAT documentation.

NAT stands for *network address translation*. See An introduction to Network Address Translation.

Cost optimization

Access to PaaS services should be through Azure Private Link or service endpoints (including storage), to avoid using a NAT gateway. Private Link and service endpoints do not require traversal of the NAT gateway to access PaaS services. This approach will reduce the charge per GB of data processed, when comparing the costs of a NAT gateway to Private Link or to service endpoints. There are additional security benefits for using Private Link or service endpoints.

Performance efficiency

Each NAT gateway resource provides up to 50 Gbps of throughput. You can split your deployments into multiple subnets, and then you can assign each subnet or groups of subnets a NAT gateway to scale out.

Each NAT gateway supports 64,000 flows for TCP and UDP respectively, per assigned outbound IP address. Up to 16 IP addresses can be assigned to a NAT gateway. The IP addresses can be individual Standard Public IP addresses, the Public IP prefix, or both. Review the following section on Source Network Address Translation (SNAT) for details. TCP stands for *Transmission Control Protocol*, and UDP stands for *User Datagram Protocol*.

SNAT exhaustion

- NAT gateway resources have a default TCP idle timeout of 4 minutes. If this setting is changed to a higher value, NAT will hold on to flows longer and can cause unnecessary pressure on SNAT port inventory.
- Atomic requests (one request per connection) are a poor design choice, because it limits scale, reduces performance, and reduces reliability. Instead, reuse HTTP/S connections, to reduce the numbers of connections and associated SNAT ports. Connection reuse will better allow the application to scale. Application performance will improve, due to reduced handshakes, overhead, and cryptographic operation costs when using TLS.
- DNS can introduce many individual flows at volume, when the client isn't caching the DNS resolvers result. Use DNS caching to reduce the volume of flows and reduce the number of SNAT ports. DNS typically stands for *Domain Name System*, the naming system for the resources that are connected to the Internet or to a private network.
- UDP flows, such as DNS lookups, use SNAT ports during the idle timeout. The longer the idle timeout, the higher the pressure on SNAT ports. A shorter idle timeout, such as 4 minutes, will reduce the length of time that the SNAT ports will be in use.
- Use connection pools to shape your connection volume.
- Never silently abandon a TCP flow and rely on TCP timers to clean up flow. If you don't let TCP explicitly close

the connection, the TCP connection remains open. Intermediate systems and endpoints will keep this connection in use, which in turn makes the SNAT port unavailable for other connections. This anti-pattern can trigger application failures and SNAT exhaustion.

- Don't change OS-level TCP-close related timer values, without expert knowledge of the implications. While the TCP stack will recover, your application performance can be negatively affected when the endpoints of a connection have mismatched expectations. Changing timer values is usually a sign of an underlying design problem. If the underlying application has other anti-patterns, SNAT exhaustion can also be amplified if the timer values are altered.

Review the following guidance to improve the scale and reliability of your service:

- Explore the effect of reducing the TCP idle timeout to lower values. A default idle timeout of 4 minutes can free up SNAT port inventory earlier.
- Consider asynchronous polling patterns for long-running operations, to free up your connection resources for other operations.
- Long-lived flows, such as reused TCP connections, should use TCP keepalives or application-layer keepalives, to avoid intermediate systems timing out. You should only increase the idle timeout as a last resort, and it might not resolve the root cause. A long timeout can cause low rate failures, when the timeout expires, and it can introduce a delay and unnecessary failures.
- Graceful retry patterns should be used to avoid aggressive retries/bursts during transient failure or failure recovery. An antipattern, called *atomic connections*, is when you create a new TCP connection for every HTTP operation. Atomic connections will prevent your application from scaling well and will waste resources. Always pipeline multiple operations into the same connection. Your application will benefit in transaction speed and resource costs. When your application uses transport layer encryption (for example, TLS), there's a significant cost associated with the processing of new connections. See Azure Cloud Design Patterns for more best-practice patterns.

Operational excellence

Although NAT gateway can be used with Azure Kubernetes Service (AKS), it isn't managed as part of AKS. If you assign a NAT gateway to the CNI subnet, you will enable AKS pods to egress through the NAT gateway.

When using multiple NAT gateways across zones or across regions, keep the outbound IP estate manageable by using Azure Public IP prefixes or BYOIP prefixes. If the IP prefix is larger than 16 IP addresses, you can create individual IP addresses from the IP Prefix and assign them to the NAT gateway.

Use Azure Monitor alerts to monitor and alert on SNAT port usage.

When a subnet is configured with a NAT gateway, the NAT gateway will replace all other outbound connectivity to the public Internet for all the VMs on that subnet. NAT gateway will take precedence over a load balancer with or without outbound rules, and over public IP addresses assigned directly to VMs. Azure tracks the direction of a flow, and asymmetric routing will not occur. Inbound originated traffic will be translated correctly, such as a load balancer frontend IP, and it will be translated separately from outbound originated traffic through a NAT gateway. This separation allows inbound and outbound services to coexist seamlessly.

NAT gateway is recommended as the default for enabling outbound connectivity for virtual networks. NAT gateway is more efficient and less operationally complex than other outbound connectivity techniques in Azure. NAT gateways allocate SNAT ports on-demand and use a more efficient algorithm to prevent SNAT port reuse conflicts. Don't rely on *default outbound connectivity* (an anti-pattern) for your estate. Instead, explicitly define it with NAT gateway resources.

Reliability

NAT gateway resources are highly available and span multiple fault domains. This is true even if a NAT gateway

is deployed regionally, without availability zones. When using availability zones for zone isolation, NAT gateways can also be deployed zonally.

Availability zone isolation cannot be provided, unless each subnet only has resources within a specific zone. Instead, deploy a subnet for each of the availability zones where VMs are deployed, align the zonal VMs with matching zonal NAT gateways, and build separate zonal stacks. For example, a virtual machine in availability zone 1 is on a subnet with other resources that are also only in availability zone 1. A NAT gateway is configured in availability zone 1 to serve that subnet. See the following diagram.

Security

A common approach is to design an outbound-only network virtual appliance (NVA) scenario with third-party firewalls or with proxy servers. When a NAT gateway is deployed to a subnet with a virtual machine scale set of NVAs, those NVAs will use the NAT gateway address(es) for outbound connectivity, as opposed to the IP of a load balancer or the individual IPs. To employ this scenario with Azure Firewall, see Integrate Azure Firewall with Azure Standard Load Balancer.

Microsoft Defender for Cloud can monitor for any suspicious outbound connectivity through a NAT gateway. This is an alert feature in Microsoft Defender for Cloud.

Next steps

- Microsoft Well-Architected Framework

- Tutorial: Create a NAT gateway using the Azure portal

Related resources

- Azure Firewall architecture overview
- Firewall and Application Gateway for virtual networks
- Multi-region load balancing with Traffic Manager and Application Gateway
- Building solutions for high availability using Availability Zones

Security design principles

11/19/2021 • 4 minutes to read • Edit Online

These principles support these three key strategies and describe a securely architected system hosted on cloud or on-premises datacenters (or a combination of both). Application of these principles will dramatically increase the likelihood your security architecture will maintain assurances of confidentiality, integrity, and availability.

Each recommendation in this document includes a description of why it is recommended, which maps to one of more of these principles:

- **Align security priorities to mission:** Security resources are almost always limited, so prioritize efforts and assurances by aligning security strategy and technical controls to the business using classification of data and systems. Security resources should be focused first on people and assets (such as systems, data, and accounts) with intrinsic business value and those with administrative privileges over business critical assets.
- **Build a comprehensive strategy:** A security strategy should consider investments in culture, processes, and security controls across all system components. The strategy should also consider security for the full lifecycle of system components including the supply chain of software, hardware, and services.
- **Drive simplicity:** Complexity in systems leads to increased human confusion, errors, automation failures, and difficulty of recovering from an issue. Favor simple and consistent architectures and implementations.
- **Design for attackers:** Your security design and prioritization should be focused on the way attackers see your environment, which is often not the way IT and application teams see it. Inform your security design and test it with **penetration testing** to simulate one-time attacks. Use **red teams** to simulate long-term persistent attack groups. Design your enterprise segmentation strategy and other security controls to **contain** attacker lateral movement within your environment. Actively measure and reduce the potential **attack surface** that attackers target for exploitation of resources within the environment.
- **Use native controls:** Favor native security controls built into cloud services over external controls from third parties. Native security controls are maintained and supported by the service provider, eliminating or reducing effort required to integrate external security tooling and update those integrations over time.
- **Use identity as primary access control:** Access to resources in cloud architectures is primarily governed by identity-based authentication and authorization for access controls. Your account control strategy should rely on identity systems for controlling access rather than relying on network controls or direct use of cryptographic keys.
- **Accountability:** Designate clear ownership of assets and security responsibilities and ensure actions are traceable for nonrepudiation. You should also ensure entities have been granted the least privilege required (to a manageable level of granularity).
- **Embrace automation:** Automation of tasks decreases the chance of human error that can create risk, so both IT operations and security best practices should be automated as much as possible to reduce human errors (while ensuring skilled humans govern and audit the automation).
- **Focus on information protection:** Intellectual property is frequently one of the biggest repositories of organizational value and this data should be protected anywhere it goes including cloud services, mobile devices, workstations, or collaboration platforms (without impeding collaboration that allows for business value creation). Your security strategy should be built around classifying information and assets

to enable security prioritization, using strong access control and encryption technology, and meeting business needs like productivity, usability, and flexibility.

- **Design for resilience:** Your security strategy should assume that controls will fail and design accordingly. Making your security posture more resilient requires several approaches working together:
  - **Balanced investment:** Invest across core functions spanning the full NIST Cybersecurity Framework lifecycle (identify, protect, detect, respond, and recover) to ensure that attackers who successfully evade preventive controls lose access from detection, response, and recovery capabilities.
  - **Ongoing maintenance:** Maintain security controls and assurances to ensure that they don't decay over time with changes to the environment or neglect.
  - **Ongoing vigilance:** Ensure that anomalies and potential threats that could pose risks to the organizations are addressed in a timely manner.
  - **Defense in depth:** Consider additional controls in the design to mitigate risk to the organization in the event a primary security control fails. This design should consider how likely the primary control is to fail, the potential organizational risk if it does, and the effectiveness of the additional control (especially in the likely cases that would cause the primary control to fail).
  - **Least privilege:** This is a form of defense in depth to limit the damage that can be done by any one account. Accounts should be granted the least amount of privilege required to accomplish their assigned tasks. Restrict the access by permission level and by time. This helps mitigate the damage of an external attacker who gains access to the account and/or an internal employee who inadvertently or deliberately (for example, insider attack) compromises security assurances.
- **Baseline and benchmark:** To ensure your organization considers current thinking from outside sources, evaluate your strategy and configuration against external references (including compliance requirements). This helps to validate your approaches, minimize risk of inadvertent oversight, and the risk of punitive fines for noncompliance.
- **Drive continuous improvement:** Systems and existing practices should be regularly evaluated and improved to ensure they are and remain effective against attackers who continuously improve and the continuous digital transformation of the enterprise. This should include processes that proactively integrate learnings from real world attacks, realistic penetration testing and red team activities, and other sources as available.
- **Assume zero trust:** When evaluating access requests, all requesting users, devices, and applications should be considered untrusted until their integrity can be sufficiently validated. Access requests should be granted conditionally based on the requestor's trust level and the target resource's sensitivity. Reasonable attempts should be made to offer means to increase trust validation (for example, request multifactor authentication) and remediate known risks (change known-leaked password, remediate malware infection) to support productivity goals.
- **Educate and incentivize security:** The humans that are designing and operating the cloud workloads are part of the whole system. It is critical to ensure that these people are educated, informed, and incentivized to support the security assurance goals of the system. This is particularly important for people with accounts granted broad administrative privileges.

Azure AD join for Azure Virtual Desktop

11/19/2021 • 7 minutes to read • Edit Online

Azure Active Directory (Azure AD) provides many benefits for organizations, such as modern authentication protocols, single sign-on (SSO), and support for FSLogix user profiles. Azure Virtual Desktop virtual machine (VM) session hosts can join directly to Azure AD. Joining directly to Azure AD removes the previous need to use Active Directory Domain Services (AD DS) domain controllers.

Originally, Azure Virtual Desktop domain join needed both Azure AD and AD DS domain controllers. Traditional Windows Server AD DS domain controllers were on-premises machines, Azure VMs, or both. Azure Virtual Desktop accessed the controllers over a site-to-site virtual private network (VPN) or Azure ExpressRoute. Alternatively, Azure Active Directory Domain Services platform-as-a-service (PaaS) provided AD DS in Azure and supported trust relationships to existing on-premises AD DS. Users had to sign in to both Azure AD and AD DS.

Other services that Azure Virtual Desktop hosts consume, such as applications and Server Message Block (SMB) storage, might still require AD DS. But Azure Virtual Desktop itself no longer requires AD DS. Removing this requirement reduces cost and complexity.

Azure AD domain join for Azure Virtual Desktop provides a modern approach for smartcards, FIDO2, authentication protocols like Windows Hello for Business, and future capabilities. Azure AD domain join also opens up the possibility of decommissioning Active Directory, since Azure Virtual Directory host pools no longer require Active Directory.

This article describes how to configure Azure AD domain join for Azure Virtual Desktop, along with some troubleshooting tips. For most Windows Azure Virtual Desktop clients, the Azure AD join configuration consists of two steps, deploying the host pool and enabling user access. For non-Windows Azure Virtual Desktop clients and other cases that need further configuration, see Protocol and client options.

Prerequisites

There are a few limitations for Azure Virtual Desktop Azure AD domain join:

- Azure AD join is only supported on Azure Virtual Desktop for Azure Resource Manager. Azure Virtual Desktop Classic isn't supported.
- Only personal host pools are currently supported. This limitation isn't in multisession pooled host pools, but in Azure Files. Azure Files currently doesn't support Azure AD as a Kerberos realm, only Active Directory. This lack of Kerberos support prevents FSLogix from working. FSLogix is the technology that manages roaming user profiles in a pooled host pool scenario.
- The session hosts must be Windows 10 Enterprise version 2004 or later.

Step 1: Deploy an Azure AD join host pool

To deploy an Azure AD host pool, follow the instructions in Create a host pool. On the **Create a host pool** screen, on the **Virtual Machines** tab, under **Domain to join**, select **Azure Active Directory**.

Selecting **Azure Active Directory** presents the option to enroll the VMs with Intune. Select **Yes** if you want to enroll the VM with Intune.

Intune can apply policies, distribute software, and help you manage VMs. For more information about Intune as part of Microsoft Endpoint Manager, see Getting started with Microsoft Endpoint Manager.

In the deployment, a new extension called **AADLoginForWindows** creates the Azure AD join and the Intune enrollment if selected.

You can also add session hosts to an existing host pool and have them Azure AD joined and Intune enrolled.

After you create the host pool VMs, you can see the VMs in **Azure AD** > **Devices**.

Name	Enabled	OS	Version	Join Type	Owner	User name	MDM
AADJ2-1	Yes	Windows	10.0.19042.928	Azure AD joined	Chris	Chris@xenithit.com	Microsoft Intune
Client	Yes	Windows	10.0.19042.928	Azure AD joined	Tom	Tom@xenithit.com	Microsoft Intune
AADJ1-0	Yes	Windows	10.0.21390.2025	Azure AD joined	None	None	Microsoft Intune

To confirm Azure AD registrations, go to **Azure Active Directory** > **Devices** > **Audit Logs** and look for **Register device**.

6/25/2021, 2:25:05 PM	Device Registration Service	Device	Register device	Success
6/28/2021, 3:48:27 PM	Device Registration Service	Device	Register device	Success
6/22/2021, 3:35:10 PM	Device Registration Service	Device	Register device	Success
6/25/2021, 2:14:49 PM	Device Registration Service	Device	Register device	Success

The VMs also appear in the MEM portal, in the **Devices** section.

Device name	Managed by	Ownership	Compliance	OS	OS version
AADJ1-0	Intune	Corporate	Compliant	Windows	10.0.21390.2025
AADJ1-1	Intune	Corporate	Compliant	Windows	10.0.22000.51
AADJ2-0	Intune	Corporate	Compliant	Windows	10.0.19042.928
AADJ2-1	Intune	Corporate	Compliant	Windows	10.0.19042.928

If a VM doesn't appear or you want to confirm enrollment, sign in to the VM locally and at a command prompt,

run the following command:

```
dsregcmd /status
```

The output shows the VM's Azure AD join status.

```
C:\>dsregcmd /status

+----------------------------------------
| Device State
+----------------------------------------

             AzureAdJoined : YES
          EnterpriseJoined : NO
              DomainJoined : NO
               Device Name : test-0

+----------------------------------------
```

On the local client, the Azure AD registration logs are in Event Viewer at **Applications and Services Logs** > **Microsoft** > **Windows** > **User Device Registration** > **Admin**.

> **NOTE**
>
> With the previous, AD DS scenario, you could manually deploy session host VMs in a separate subscription connected to a different Azure AD if necessary. The VMs had no dependency on Azure AD. The VMs only needed network line of sight to an AD DS domain controller in a domain that synchronized user objects to the Azure Virtual Desktops' Azure AD.
>
> Azure AD join doesn't support this scenario. The host VMs automatically join to the Azure AD of the subscription that deploys the VMs. The deployment inherits that Azure AD as an identity provider, and uses the user identities that the Azure AD holds. There's no way to specify a different Azure AD for the host VMs. So be sure to create the VMs in the same subscription as all the other Azure Virtual Desktop objects. The VMs also automatically enroll into the Intune tenant associated with the Azure AD.

Step 2: Enable user access

In the next step, you enable sign-in access to the VMs. These VMs are Azure objects, and the authentication mechanism is Azure AD. You manage user sign-in permission through Azure role-based access control (RBAC).

In Azure Virtual Desktop, users must be in the Azure Virtual Desktop Desktop application group to sign in to the VMs. For Azure AD join, the same users and groups that are in the Desktop application group must also be added to the **Virtual Machine User Login** RBAC role. This role isn't a Azure Virtual Desktop role, but an Azure role with the **Log in to Virtual Machine** DataAction permission.

Choose the scope for this role.

- Assigning the role at the **VM level** means you have to assign the role for each VM you add.

- Assigning the role at the **resource group level** means the role automatically applies to all VMs in that resource group.
- Assigning the role at the **Subscription level** means users can sign in to all VMs in the subscription.

Setting the role once at the resource group level might be the best option. This approach prevents having to assign the role for every VM, but avoids assigning it at the top level of the subscription.

To assign the **Virtual Machine User Login** role:

1. In the Azure portal, go to your chosen scope, for example the resource group, and select **Access control (IAM)**.

2. At the top of the screen, select **+ Add** > **Add role assignment**.
3. Under **Role**, select **Virtual Machine User Login**, and under **Select**, select the same user group that's assigned to the Desktop Application Group.

The user group now appears under **Virtual Machine User Login**.

If you don't assign this role, users get an error message when they try to sign in via the Windows client.

Web client users get a different-looking error.

Local Admin access

To give a user local administrative access to the VM, also add the user to the **Virtual Machine Administrator Login** role. This role has a **Log in to Virtual Machine as administrator** DataAction permission that enables administrative access.

Protocol and client options

By default, host pool access only works from the Windows Azure Virtual Desktop client. To access host pool VMs, your local computer must be:

- Azure AD-joined or hybrid Azure AD-joined to the same Azure AD tenant as the session host.
- Running Windows 10 version 2004 or later, and also Azure AD-registered to the same Azure AD tenant as the session host.

Host pool access uses the Public Key User to User (PKU2U) protocol for authentication. To sign in to the VM, the session host and the local computer must have the PKU2U protocol enabled. For Windows 10 version 2004 or later machines, if the PKU2U protocol is disabled, enable it in the Windows registry as follows:

1. Navigate to **HKLM\SYSTEM\CurrentControlSet\Control\Lsa\pku2u**.
2. Set **AllowOnlineID** to **1**.

If your client computers use Group Policy, also enable the Group Policy Option:

1. Navigate to **Computer Configuration\Policies\Windows Settings\Security Settings\Local Policies\Security Options**.
2. Under **Policy**, set **Network security: Allow PKU2U authentication requests to this computer to use online identities** to **Enabled**.

If you're using other Azure Virtual Desktop clients, such as Mac, iOS, Android, web, the Store client, or pre-version 2004 Windows 10, enable the RDSTLS protocol. Enable this protocol by adding a new custom RDP Property to the host pool, *targetisaadjoined:i:1*. Azure Virtual Desktop then uses this protocol instead of PKU2U.

Now you have an Azure Virtual Desktop host pool where the session hosts are joined only to Azure AD. You're a step closer to modern management for your Azure Virtual Desktop estate.

Next steps

- Azure Virtual Desktop documentation
- Deploy Azure AD-joined virtual machines in Azure Virtual Desktop

Related resources

- Azure Virtual Desktop for the enterprise
- Integrate on-premises AD domains with Azure AD

Ten design principles for Azure applications

11/19/2021 • 2 minutes to read • Edit Online

Follow these design principles to make your application more scalable, resilient, and manageable.

Design for self healing. In a distributed system, failures happen. Design your application to be self healing when failures occur.

Make all things redundant. Build redundancy into your application, to avoid having single points of failure.

Minimize coordination. Minimize coordination between application services to achieve scalability.

Design to scale out. Design your application so that it can scale horizontally, adding or removing new instances as demand requires.

Partition around limits. Use partitioning to work around database, network, and compute limits.

Design for operations. Design your application so that the operations team has the tools they need.

Use managed services. When possible, use platform as a service (PaaS) rather than infrastructure as a service (IaaS).

Use the best data store for the job. Pick the storage technology that is the best fit for your data and how it will be used.

Design for evolution. All successful applications change over time. An evolutionary design is key for continuous innovation.

Build for the needs of business. Every design decision must be justified by a business requirement.

Design and implementation patterns

11/19/2021 • 2 minutes to read • Edit Online

Good design encompasses factors such as consistency and coherence in component design and deployment, maintainability to simplify administration and development, and reusability to allow components and subsystems to be used in other applications and in other scenarios. Decisions made during the design and implementation phase have a huge impact on the quality and the total cost of ownership of cloud hosted applications and services.

PATTERN	SUMMARY
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Anti-Corruption Layer	Implement a façade or adapter layer between a modern application and a legacy system.
Backends for Frontends	Create separate backend services to be consumed by specific frontend applications or interfaces.
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Compute Resource Consolidation	Consolidate multiple tasks or operations into a single computational unit
External Configuration Store	Move configuration information out of the application deployment package to a centralized location.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Offloading	Offload shared or specialized service functionality to a gateway proxy.
Gateway Routing	Route requests to multiple services using a single endpoint.
Leader Election	Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances.
Pipes and Filters	Break down a task that performs complex processing into a series of separate elements that can be reused.
Sidecar	Deploy components of an application into a separate process or container to provide isolation and encapsulation.
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.

PATTERN	SUMMARY
Strangler Fig	Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services.